

Oracle Rdb™

SQL Reference Manual
Volume 3

Oracle Rdb Release 7.1.4.3 for OpenVMS Alpha
December 2005

ORACLE®

SQL Reference Manual, Volume 3

Oracle Rdb Release 7.1.4.3 for OpenVMS Alpha

Copyright © 1987, 2005 Oracle Corporation. **All rights reserved.** All rights reserved.

The Programs (which include both the software and documentation) contain proprietary information of Oracle Corporation; they are provided under a license agreement containing restrictions on use and disclosure and are also protected by copyright, patent and other intellectual and industrial property laws. Reverse engineering, disassembly, or decompilation of the Programs is prohibited.

The information contained in this document is subject to change without notice. If you find any problems in the documentation, please report them to us in writing. Oracle Corporation does not warrant that this document is error-free. Except as may be expressly permitted in your license agreement for these Programs, no part of these Programs may be reproduced or transmitted in any form or by any means, electronic or mechanical, for any purpose, without the express written permission of Oracle Corporation.

If the Programs are delivered to the U.S. Government or anyone licensing or using the programs on behalf of the U.S. Government, the following notice is applicable:

U.S. GOVERNMENT RIGHTS Programs, software, databases, and related documentation and technical data delivered to U.S. Government customers are "commercial computer software" or "commercial technical data" pursuant to the applicable Federal Acquisition Regulation and agency-specific supplemental regulations. As such, use, duplication, disclosure, modification, and adaptation of the Programs, including documentation and technical data, shall be subject to the licensing restrictions set forth in the applicable Oracle license agreement, and, to the extent applicable, the additional rights set forth in FAR 52.227-19, Commercial Computer Software - Restricted Rights (June, 1987). Oracle Corporation, 500 Oracle Parkway, Redwood City, CA 94065.

The Programs are not intended for use in any nuclear, aviation, mass transit, medical, or other inherently dangerous applications. It shall be the licensee's responsibility to take all appropriate fail-safe, backup, redundancy, and other measures to ensure the safe use of such applications if the Programs are used for such purposes, and Oracle Corporation disclaims liability for any damages caused by such use of the Programs.

Oracle is a registered trademark, and Hot Standby, LogMiner for Rdb, Oracle CDD/Repository, Oracle CODASYL DBMS, Oracle Rdb, Oracle RMU, Oracle SQL/Services, Oracle Trace, and Rdb7 are trademark or registered trademarks of Oracle Corporation. Other names may be trademarks of their respective owners.

The Programs may provide links to Web sites and access to content, products, and services from third parties. Oracle is not responsible for the availability of, or any content provided on, third-party Web sites. You bear all risks associated with the use of such content. If you choose to purchase any products or services from a third party, the relationship is directly between you and the third party. Oracle is not responsible for: (a) the quality of third-party products or services; or (b) fulfilling any of the terms of the agreement with the third party, including delivery of products or services and warranty obligations related to purchased products or services. Oracle is not responsible for any loss or damage of any sort that you may incur from dealing with any third party.

Contents

Send Us Your Comments	xi
Preface	xiii
7 SQL Statements	
CREATE SEQUENCE Statement	7-2
CREATE STORAGE AREA Clause	7-11
CREATE STORAGE MAP Statement	7-25
CREATE SYNONYM Statement	7-52
CREATE TABLE Statement	7-56
CREATE TRIGGER Statement	7-109
CREATE USER Statement	7-130
CREATE VIEW Statement	7-133
DECLARE ALIAS Statement	7-143
DECLARE CURSOR Statement	7-155
DECLARE CURSOR Statement, Dynamic	7-172
DECLARE CURSOR Statement, Extended Dynamic	7-178
DECLARE FUNCTION Statement	7-185
DECLARE LOCAL TEMPORARY TABLE Statement	7-186
DECLARE MODULE Statement	7-195
DECLARE PROCEDURE Statement	7-205
DECLARE Routine Statement	7-206
DECLARE STATEMENT Statement	7-213
DECLARE TABLE Statement	7-215
DECLARE TRANSACTION Statement	7-222
DECLARE Variable Statement	7-233
DELETE Statement	7-236
DESCRIBE Statement	7-242

DISCONNECT Statement	7-249
DROP Statements	7-254
DROP CATALOG Statement	7-255
DROP COLLATING SEQUENCE Statement	7-258
DROP CONSTRAINT Statement	7-261
DROP DATABASE Statement	7-263
DROP DOMAIN Statement	7-266
DROP INDEX Statement	7-269
DROP MODULE Statement	7-271
DROP OUTLINE Statement	7-274
DROP PATHNAME Statement	7-276
DROP PROFILE Statement	7-277
DROP ROLE Statement	7-280
Drop Routine Statement	7-282
DROP SCHEMA Statement	7-286
DROP SEQUENCE Statement	7-289
DROP STORAGE MAP Statement	7-292
DROP SYNONYM Statement	7-294
DROP TABLE Statement	7-296
DROP TRIGGER Statement	7-301
DROP USER Statement	7-303
DROP VIEW Statement	7-305
EDIT Statement	7-308
END DECLARE Statement	7-312
Execute (@) Statement	7-315
EXECUTE Statement	7-318
EXECUTE IMMEDIATE Statement	7-326
EXIT Statement	7-331
EXPORT Statement	7-332
FETCH Statement	7-337
FOR Control Statement	7-346
FOR (Counted) Control Statement	7-350
GET DIAGNOSTICS Statement	7-357
GET ENVIRONMENT Statement	7-366
GRANT Statements	7-369
GRANT Statement	7-371
GRANT Statement: ANSI/ISO-Style	7-392

GRANT Statement: Roles	7-404
------------------------------	-------

Index

Tables

7-1	Using Temporary Tables	7-82
7-2	Availability of Row Data for Triggered Actions	7-115
7-3	Classes, Types, and Modes of Cursors	7-157
7-4	GET ENVIRONMENT session keywords	7-367
7-5	SQL Privileges for Databases, Tables, Columns, Modules, External Routines and Sequences	7-377
7-6	Privilege Override Capability	7-384

Send Us Your Comments

Oracle Rdb for OpenVMS Oracle SQL Reference Manual, Release 7.1.4.1

Oracle Corporation welcomes your comments and suggestions on the quality and usefulness of this publication. Your input is an important part of the information used for revision.

- Did you find any errors?
- Is the information clearly presented?
- Do you need more information? If so, where?
- Are the examples correct? Do you need more examples?
- What features did you like most?

If you find any errors or have any other suggestions for improvement, please indicate the document title, chapter, section, and page number (if available). You can send comments to us in the following ways:

- Electronic mail: nedc-doc_us@oracle.com
- FAX — 603-897-3825 Attn: Oracle Rdb
- Postal service:
Oracle Corporation
Oracle Rdb Documentation
One Oracle Drive
Nashua, NH 03062-2804
USA

If you would like a reply, please give your name, address, telephone number, and (optionally) electronic mail address.

If you have problems with the software, please contact your local Oracle Support Services.

Preface

This manual describes the syntax and semantics of the statements and language elements for the SQL (structured query language) interface to the Oracle Rdb database software.

Intended Audience

To get the most out of this manual, you should be familiar with data processing procedures, basic database management concepts and terminology, and the OpenVMS operating system.

Operating System Information

You can find information about the versions of the operating system and optional software that are compatible with this version of Oracle Rdb in the *Oracle Rdb Installation and Configuration Guide*.

For information on the compatibility of other software products with this version of Oracle Rdb, refer to the *Oracle Rdb Release Notes*.

Contact your Oracle representative if you have questions about the compatibility of other software products with this version of Oracle Rdb.

Structure

This manual is divided into five volumes. Volume 1 contains Chapter 1 through Chapter 5 and an index. Volume 2 contains Chapter 6 and an index. Volume 3 contains Chapter 7 and an index. Volume 4 contains Chapter 8 and an index. Volume 5 contains the appendixes and an index.

The index for each volume contains entries for the respective volume only and does not contain index entries from the other volumes in the set.

The following table shows the contents of the chapters and appendixes in Volumes 1, 2, 3, 4, and 5 of the *Oracle Rdb SQL Reference Manual*:

Chapter 1	Introduces SQL (structured query language) and briefly describes SQL functions. This chapter also describes conformance to the ANSI standard, how to read syntax diagrams, executable and nonexecutable statements, keywords and line terminators, and support for Multivendor Integration Architecture.
Chapter 2	Describes the language and syntax elements common to many SQL statements.
Chapter 3	Describes the syntax for the SQL module language and the SQL module processor command line.
Chapter 4	Describes the syntax of the SQL precompiler command line.
Chapter 5	Describes SQL routines.
Chapter 6 Chapter 7 Chapter 8	Describe in detail the syntax and semantics of the SQL statements. These chapters include descriptions of data definition statements, data manipulation statements, and interactive control commands.
Appendix A	Describes the different types of errors encountered in SQL and where they are documented.
Appendix B	Describes the SQL standards to which Oracle Rdb conforms.
Appendix C	Describes the SQL Communications Area, the message vector, and the SQLSTATE error handling mechanism.
Appendix D	Describes the SQL Descriptor Areas and how they are used in dynamic SQL programs.
Appendix E	Summarizes the logical names that SQL recognizes for special purposes.
Appendix F	Summarizes the obsolete SQL features of the current Oracle Rdb version.
Appendix G	Summarizes the SQL functions that have been added to the Oracle Rdb SQL interface for convergence with Oracle7 SQL. This appendix also describes the SQL syntax for performing an outer join between tables.
Appendix H	Describes information tables that can be used with Oracle Rdb.
Index	Index for each volume.

Related Manuals

For more information on Oracle Rdb, see the other manuals in this documentation set, especially the following:

- *Oracle Rdb Guide to Database Design and Definition*
- *Oracle Rdb7 Guide to Database Performance and Tuning*
- *Oracle Rdb Introduction to SQL*
- *Oracle Rdb Guide to SQL Programming*

Conventions

In examples, an implied carriage return occurs at the end of each line, unless otherwise noted. You must press the Return key at the end of a line of input.

Often in examples the prompts are not shown. Generally, they are shown where it is important to depict an interactive sequence exactly; otherwise, they are omitted.

The following conventions are also used in this manual:

.	Vertical ellipsis points in an example mean that information not directly related to the example has been omitted.
:	
:	
...	Horizontal ellipsis points in statements or commands mean that parts of the statement or command not directly related to the example have been omitted.
e, f, t	Index entries in the printed manual may have a lowercase e, f, or t following the page number; the e, f, or t is a reference to the example, figure, or table, respectively, on that page.
boldface text	Boldface type in text indicates a new term.
< >	Angle brackets enclose user-supplied names in syntax diagrams.
[]	Brackets enclose optional clauses from which you can choose one or none.
\$	The dollar sign represents the command language prompt. This symbol indicates that the command language interpreter is ready for input.

References to Products

The Oracle Rdb documentation set to which this manual belongs often refers to the following Oracle Corporation products by their abbreviated names:

- In this manual, Oracle Rdb refers to Oracle Rdb for OpenVMS. Version 7.1 of Oracle Rdb software is often referred to as V7.1.
- Oracle CDD/Repository software is referred to as the dictionary, the data dictionary, or the repository.
- Oracle ODBC Driver for Rdb software is referred to as the ODBC driver.
- OpenVMS means the OpenVMS Alpha operating system.

7

SQL Statements

This chapter describes the syntax and semantics of statements in SQL. SQL statements include data definition statements; data manipulation statements; statements that control the environment and program flow; and statements that give information.

See Chapter 2 in Volume 1 for detailed descriptions of the language and syntax elements referred to by the syntax diagrams in this chapter.

Chapter 6 in Volume 2 describes the statements from `ACCEPT` to `CREATE SCHEMA`.

Chapter 8 in Volume 4 describes the statements from `HELP` to `WHILE`.

CREATE SEQUENCE Statement

CREATE SEQUENCE Statement

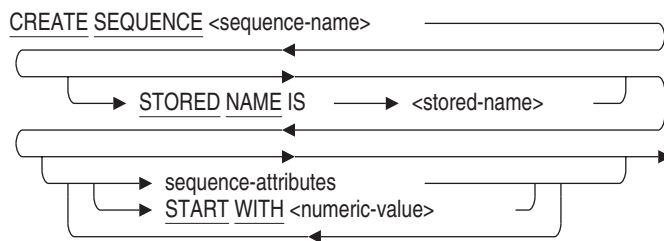
Creates a sequence. A sequence is a database object from which multiple users can generate unique integers. You can use sequences to automatically generate primary key values.

Environment

You can use the CREATE SEQUENCE statement:

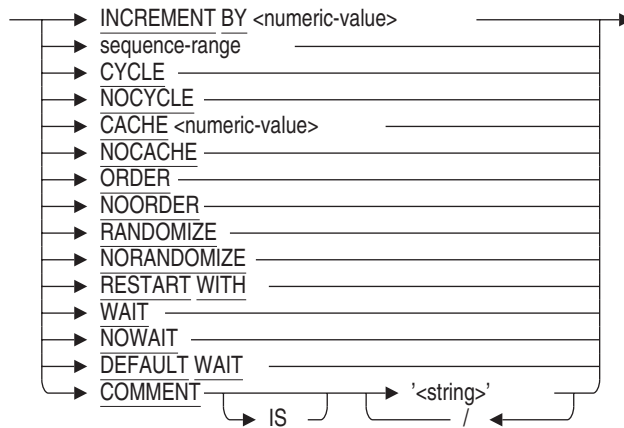
- In interactive SQL
- Embedded in host language programs
- As part of a procedure in an SQL module or other compound statement
- In dynamic SQL as a statement to be dynamically executed

Format

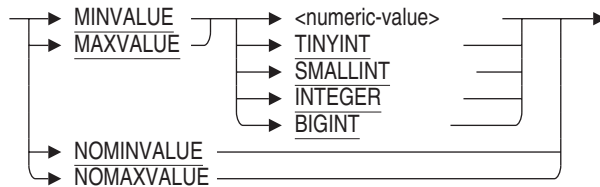


CREATE SEQUENCE Statement

sequence-attributes =



sequence-range =



Arguments

CACHE numeric-value

NOCACHE

The CACHE clause specifies how many values of the sequence Oracle Rdb should preallocate and keep in memory for faster access. The numeric value must be between 2 and 2147483647.

You cannot cache more values than will fit in a given cycle of sequence numbers; thus, the maximum value allowed for the CACHE clause must be less than the value resulting from the following formula:

$$(\text{MAXVALUE} - \text{MINVALUE}) / \text{ABS}(\text{INCREMENT})$$

The SET FLAGS option SEQ_CACHE can be used to override the setting of CACHE at runtime. See the SET FLAGS Statement for more details.

CREATE SEQUENCE Statement

A cache for a given sequence is populated at the first request for a number from that sequence, and whenever a value is requested when the cache is empty. If a system failure occurs, or when the cache is released any unfetched values will be discarded. The maximum number of lost values is equal to the current cache size. This may be the value specified by `CACHE` or by the `SET FLAGS SEQ_CACHE` option.

The `NOCACHE` clause specifies that values will be allocated one at a time. This will require more I/O to the Rdb root file than using a `CACHE` value.

By default, Oracle Rdb caches 20 sequence values.

COMMENT IS 'string '

Adds a comment about the sequence. SQL displays the text of the comment when it executes a `SHOW SEQUENCE` statement. Enclose the comment in single quotation marks (') and separate multiple lines in a comment with a slash mark (/).

CYCLE

NOCYCLE

The `CYCLE` clause specifies that the sequence is to continue generating values after reaching either the `MINVALUE` or `MAXVALUE`. After an ascending sequence reaches the `MAXVALUE`, the sequence starts again from its `MINVALUE`. After a descending sequence reaches its `MINVALUE`, the sequence starts again at its `MAXVALUE`. The `NOCYCLE` clause specifies that the sequence should not continue generating values after reaching either its minimum or maximum value. An error is generated if an attempt is made to increment the sequence beyond its limits. The `NOCYCLE` clause is the default.

INCREMENT BY numeric-value

Specifies the size of the increment and the direction (ascending or descending) of the sequence. This numeric value must be in the range -2147483648 through 2147483647, excluding 0. The absolute value must be less than the difference of `MAXVALUE` and `MINVALUE`. A negative value specifies a descending sequence; a positive value specifies an ascending sequence. By default, the numeric value is 1.

MAXVALUE numeric-value

NOMAXVALUE

The `MAXVALUE` clause specifies the maximum signed quadword (`BIGINT`) value that the sequence can generate. The numeric value must be between -9223372036854775808 and 9223372036854775808. The `MAXVALUE` must be equal to or greater than the value specified for the `START WITH` clause and greater than the value specified with the `MINVALUE` clause. The `NOMAXVALUE` clause specifies that the maximum value for an ascending

CREATE SEQUENCE Statement

sequence is 9223372036854775808 (plus the cache size) and -1 for a descending sequence.

The NOMAXVALUE clause is the default.

MAXVALUE TINYINT
MAXVALUE SMALLINT
MAXVALUE INTEGER
MAXVALUE BIGINT

SQL allows the keyword TINYINT, SMALLINT, INTEGER and BIGINT to follow MAXVALUE instead of a numeric value. This allows easy range setting for sequences used with these data types. The value supplied will be the largest positive value that can be assigned to this data type.

MINVALUE numeric-value
NOMINVALUE

The MINVALUE clause specifies the minimum signed quadword (BIGINT) value that the sequence can generate. The numeric value must be equal to or greater than -9223372036854775808. The MINVALUE must be less than or equal to the value specified with the START WITH clause and less than the value specified with the MAXVALUE clause. The NOMINVALUE clause specifies that the minimum value for an ascending sequence is 1, and -9223372036854775808 (plus the cache size) for a descending sequence.

The NOMINVALUE clause is the default.

MINVALUE TINYINT
MINVALUE SMALLINT
MINVALUE INTEGER
MINVALUE BIGINT

SQL allows the keyword TINYINT, SMALLINT, INTEGER and BIGINT to follow MINVALUE instead of a numeric value. This allows easy range setting for sequences used with these data types. The value supplied will be the smallest negative value that can be assigned to this data type.

ORDER
NOORDER

The ORDER clause specifies that sequence numbers are guaranteed to be assigned in order for each requesting process, thus maintaining a strict history of requests. The NOORDER clause specifies that sequence numbers are not guaranteed to be generated in order of request.

The NOORDER clause is the default.

CREATE SEQUENCE Statement

RANDOMIZE NORANDOMIZE

The **RANDOMIZE** clause specifies that the sequence numbers are to be returned with a random value in the most significant bytes of the **BIGINT** value. This allows unique values to be generated that have a random distribution. When you specify the **NORANDOMIZE** clause, sequence numbers are close in value to others created at the same time.

The advantage of the **RANDOMIZE** clause is that updates to columns of a sorted index to which these values are written occur in different locations in the index structure. This may improve concurrent access for large indexes as leaf nodes in different parts of the index can be updated independently. In contrast, the sequence numbers generated when you specify the **NORANDOMIZE** clause (which are likely to be close in numeric value to other sequences) result in index updates that occur in the same or nearby index nodes, which may lead to contention in one part of the sorted index.

The full range of values in the **BIGINT** value returned for the sequence are used; therefore, the **NOMAXVALUE** and **NOMINVALUE** clauses must be specified (or defaulted to) for the sequence definition. The most significant bits of the **BIGINT** value are set to a randomly generated positive value. A generated distinct value is returned in the least significant 32 bits so that uniqueness is guaranteed. If you also specify the **CYCLE** clause, then only the least significant 32 bits are cycled. When a query is performed on the column **RDB\$NEXT_SEQUENCE_VALUE** in the **RDB\$SEQUENCES** table, only the generated value of the least significant bits is returned, because the most significant bits are not assigned until the **NEXTVAL** pseudocolumn is referenced.

If you specify **RANDOMIZE**, you cannot also specify **ORDER**, **MAXVALUE**, or **MINVALUE**. The **NORANDOMIZE** clause is the default.

sequence-name

The name of the sequence that you want to create. Use a name that is unique among all sequence names in the database, or in the schema if you are using a multischema database. Use any valid SQL name.

START WITH numeric-value

Specifies the initial numeric value to be used for the sequence. This value must be in the range specified by (or defaulted to) the other sequence attribute clauses. Valid values are in the range -9223373036854775808 to 9223372036854775807.

If omitted, the **START WITH** value defaults to the value of **MINVALUE** for ascending sequences and **MAXVALUE** for descending sequences.

CREATE SEQUENCE Statement

STORED NAME IS stored-name

Specifies a name that Oracle Rdb uses to access a sequence created in a multischema database. The stored name allows you to access multischema definitions using interfaces, such as Oracle RMU, that do not recognize multiple schemas in one database. You cannot specify a stored name for a sequence in a database that does not allow multiple schemas.

WAIT

NOWAIT

DEFAULT WAIT

Specifies what wait state is used when a reference to NEXTVAL is used. A reference to NEXTVAL for a sequence may require synchronization with other users of the sequence. When you specify DEFAULT WAIT, the wait state (WAIT or NOWAIT) of the current transaction is used. This may mean that no waiting is performed during a NOWAIT transaction.

If you specify WAIT (the default) for the sequence, then regardless of the wait state set for the current transaction, all synchronization waits for the next value. This is the recommended setting if the application uses NOWAIT transactions. The current WAIT timeout interval defined for the transaction or database is used.

If you specify NOWAIT for the sequence, then regardless of the current transaction setting, all synchronization does not wait for the next value.

Usage Notes

- You must have the CREATE database privilege on the database to create a sequence.
A user must have SELECT privileges on a sequence to use the NEXTVAL and CURRVAL pseudocolumns.
- A user must refer to the NEXTVAL pseudocolumn before he or she can use the CURRVAL pseudocolumn.
- Concurrent access is allowed to the sequence once the transaction in which the sequences were created is committed.
- If you specify the NEXTVAL pseudocolumn more than once in a statement, then only the first specification increments the sequence value; the others act as CURRVAL references.
- NEXTVAL and CURRVAL may be delimited. All upper and lower case variations of these keywords are accepted and assumed to be equivalent to these upper case keywords.

CREATE SEQUENCE Statement

The following example shows that any case is accepted.

```
SQL> set dialect 'sql99';
SQL> create sequence dept_id;
SQL> select dept_id.nextval from rdb$database;

          1
1 row selected
SQL> select "DEPT_ID".currval from rdb$database;

          1
1 row selected
SQL> select "DEPT_ID"."CURRVAL" from rdb$database;

          1
1 row selected
SQL> select "DEPT_ID"."nextval" from rdb$database;

          2
1 row selected
SQL> select "DEPT_ID"."CuRrVaL" from rdb$database;

          2
1 row selected
```

- A run-time lock is used to synchronize access to the next unused sequence value.
- The value of the `START WITH` clause establishes the initial value generated after a sequence is created. This value is not necessarily the value to which an ascending cycling sequence cycles after reaching its maximum or minimum value.
- If you specify none of the sequence attributes, an ascending sequence is created that starts with 1, increases by 1, and has no upper limit. If the only sequence attribute that you specify is `INCREMENT BY -1`, a descending sequence that starts with -1 and decreases with no lower limit is generated.
- To create a sequence that increments without bounds, do one of the following:
 - For an ascending sequence, omit the `MAXVALUE` clause or specify the `NOMAXVALUE` clause.
 - For a descending sequence, omit the `MINVALUE` clause or specify the `NOMINVALUE` clause.
- To create a sequence that stops at a predefined limit, do one of the following:
 - For an ascending sequence, specify a value for the `MAXVALUE` clause and omit the `CYCLE` clause.

CREATE SEQUENCE Statement

- For a descending sequence, specify a value for the MINVALUE clause and omit the CYCLE clause.
Any attempt to generate a sequence number once the sequence has reached its limit results in an error.
- To create a sequence that restarts after reaching a predefined limit, omit the CYCLE clause and specify values for both the MAXVALUE and MINVALUE clauses.
- Once a cache is created, you can access its values in SQL statements with the following pseudocolumns:
 - CURRVAL: Returns the current value of the sequence.
 - NEXTVAL: Increments the sequence and returns the new value.

Examples

Example 1: Creating a Sequence

```
SQL> -- This example creates a new sequence using the default
SQL> -- values for NOMINVALUE, NOMAXVALUE, INCREMENT BY 1, NOCYCLE,
SQL> -- and CACHE 20. The START WITH value is set to 147.
SQL> -- Allyn Stuart will be assigned an EMPLOYEE_ID value of 147.
SQL> -- Nick Jones will be assigned an EMPLOYEE_ID of 148.
SQL> --
SQL> CREATE SEQUENCE EMPID START WITH 00147;
SQL> -- Use NEXTVAL to fetch a sequence number for the primary key column.
SQL> INSERT INTO EMPLOYEES
cont> (EMPLOYEE_ID, LAST_NAME, FIRST_NAME)
cont> VALUES (EMPID.NEXTVAL, 'STUART', 'ALLYN')
cont> RETURNING EMPLOYEE_ID;
EMPLOYEE_ID
147
1 row inserted
SQL> -- Use CURRVAL to reuse the EMPLOYEE_ID value for the foreign key columns
SQL> -- in the associated tables.
SQL> INSERT INTO SALARY_HISTORY
cont> (EMPLOYEE_ID, SALARY_AMOUNT, SALARY_START, SALARY_END)
cont> VALUES (EMPID.CURRVAL, 35000, '6-FEB-1998', NULL)
cont> RETURNING EMPLOYEE_ID;
EMPLOYEE_ID
147
1 row inserted
SQL> INSERT INTO JOB_HISTORY
cont> (EMPLOYEE_ID, DEPARTMENT_CODE, JOB_START, JOB_END)
cont> VALUES (EMPID.CURRVAL, 'ENGR', '6-FEB-1998', NULL)
cont> RETURNING EMPLOYEE_ID;
EMPLOYEE_ID
147
```

CREATE SEQUENCE Statement

```
1 row inserted
SQL> INSERT INTO EMPLOYEES
cont> (EMPLOYEE_ID, LAST_NAME, FIRST_NAME)
cont> VALUES (EMPID.NEXTVAL, 'JONES ', 'NICK ')
cont> RETURNING EMPLOYEE_ID;
  EMPLOYEE_ID
          148
1 row inserted
```

CREATE STORAGE AREA Clause

CREATE STORAGE AREA Clause

Note

You cannot issue `CREATE STORAGE AREA` as an independent statement. It is a clause allowed only as part of a `CREATE DATABASE` or `IMPORT` statement.

You can also create a storage area using the `ADD STORAGE AREA` clause of the `ALTER DATABASE` statement.

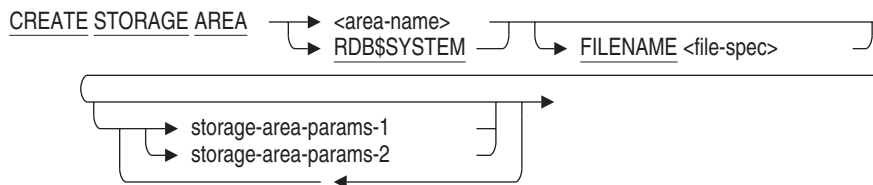
Creates additional storage areas in a multfile database. **Storage areas** are data and snapshot files that are associated with particular tables in a multfile database.

A `CREATE STORAGE AREA` clause specifies the names for the storage area files and determines their physical characteristics. Subsequent `CREATE STORAGE MAP` statements associate the storage area with particular tables in the database.

Environment

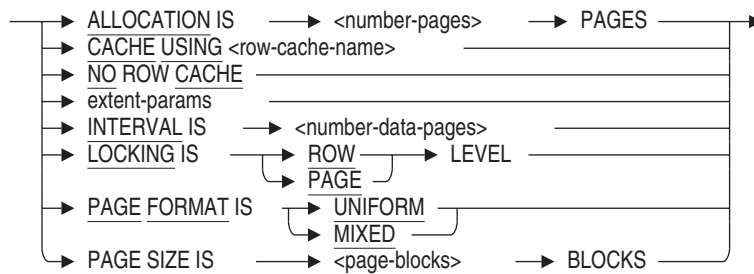
You can use the `CREATE STORAGE AREA` clause only within a `CREATE DATABASE` or `IMPORT` statement.

Format

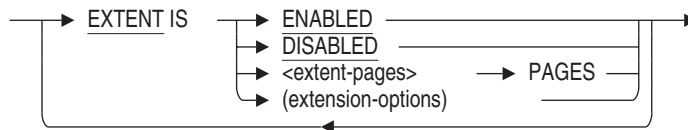


CREATE STORAGE AREA Clause

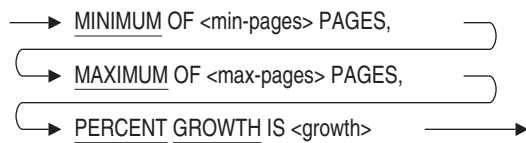
storage-area-params-1 =



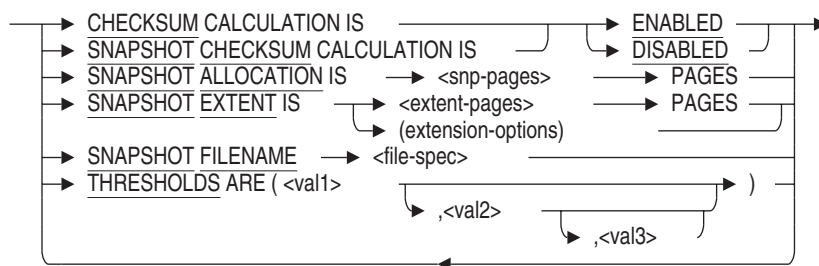
extent-params =



extension-options =



storage-area-params-2 =



CREATE STORAGE AREA Clause

Arguments

ALLOCATION IS number-pages PAGES

The number of database pages initially allocated to the storage area. Rdb will automatically extend this allocation to account for internal structure pages, such as SPAM (spage management) pages. For example, an allocation of 25 will be increased to 27 as shown in this example:

```
SQL> alter database filename MF_PERSONNEL
cont> add storage area DOC_EXAMPLE
cont>   page format is uniform
cont>   allocation 25;
SQL> attach 'filename MF_PERSONNEL';
SQL> show storage area DOC_EXAMPLE

DOC_EXAMPLE
Access is:      Read write
Page Format:    Uniform
Page Size:     2 blocks
Area File:     USER_DISK:[DOC.DATABASES]DOC_EXAMPLE.RDA;1
Area Allocation: 27 pages
Extent:        Enabled
Area Extent Minimum: 99 pages
Area Extent Maximum: 9999 pages
Area Extent Percent: 20 percent
Snapshot File: USER_DISK:[DOC.DATABASES]DOC_EXAMPLE.SNP;1
Snapshot Allocation: 100 pages
Snapshot Extent Minimum: 99 pages
Snapshot Extent Maximum: 9999 pages
Snapshot Extent Percent: 20 percent
Locking is Row Level
No Cache Associated with Storage Area
No database objects use Storage Area DOC_EXAMPLE
```

CACHE USING row-cache-name

Assigns the named row cache to the specified storage area. All rows stored in this area, whether they consist of table data, segmented string data, or special rows such as index nodes, are cached if those rows fit in the cache.

If the row cache does not exist, you must create the row cache before terminating the CREATE DATABASE statement. For example:

```
SQL> CREATE DATABASE FILENAME test_db
cont> ROW CACHE IS ENABLED
cont> CREATE STORAGE AREA area1
cont>   CACHE USING test1
cont> CREATE CACHE test1
cont>   CACHE SIZE IS 100 ROWS
cont>   ROW LENGTH IS 200 BYTES;
```

CREATE STORAGE AREA Clause

Only one row cache is allowed for each storage area.

NO ROW CACHE is the default for a storage area.

CHECKSUM CALCULATION

SNAPSHOT CHECKSUM CALCULATION

This option allows you to enable or disable calculations of page checksums when pages are read from or written to the storage or snapshot area files.

The default is ENABLED.

Note

Oracle Corporation recommends that you leave checksum calculations enabled; which is the default.

With current technology, it is possible that errors may occur that the checksum calculation can detect but that may not be detected by either the hardware, firmware, or software. Unexpected application results and database corruption may occur if corrupt pages exist in memory or on disk but are not detected.

Oracle Corporation recommends performing checksum calculations, except in the following specific circumstances:

- Your application is stable and has run without errors on the current hardware and software configuration for an extended period of time.
- You have reached maximum CPU utilization in your current configuration. Actual CPU utilization by the checksum calculation depends primarily on the size of the database pages in your database. The larger the database page, the more noticeable the CPU usage by the checksum calculation may become.

Note

Oracle Corporation recommends that you carefully evaluate the trade-off between reducing CPU usage by the checksum calculation and the potential for loss of database integrity if checksum calculations are disabled.

Oracle Rdb allows you to disable and, subsequently, re-enable checksum calculation without error. However, once checksum calculations have been disabled, corrupt pages may not be detected even if checksum calculations are subsequently re-enabled.

CREATE STORAGE AREA Clause

EXTENT ENABLED EXTENT DISABLED

Enables or disables extents. Extents are **ENABLED** by default.

You may encounter performance problems when creating hashed indexes in storage areas with the mixed page format if the storage area was created specifying the incorrect size for the area and if extents are enabled. By disabling extents, this problem can be diagnosed early and corrected to improve performance.

EXTENT IS extent-pages PAGES EXTENT IS (extension-options)

Specifies the number of pages of each storage area file extent. See also the description under the **SNAPSHOT EXTENT** argument.

FILENAME file-spec

Provides an explicit file specification for storage area files. The **CREATE STORAGE AREA** clause creates two files: a storage area file with a file extension of **.rda**, and a snapshot file with a file extension of **.snp**. If you omit the **FILENAME** argument, the file specification takes the following defaults:

- Device: the current device for the process
- Directory: the current directory for the process
- File name: the name specified for the storage area

The file specification is used for both the storage area and snapshot files that comprise the storage area (unless you use the **SNAPSHOT FILENAME** argument to specify a different file for the snapshot file). Because the **CREATE STORAGE AREA** clause can create two files with different file extensions, do not specify a file extension with the file specification.

You may use a logical name for all or part of a file specification.

One benefit of a multfile database is that its files can reside on more than one disk. If you want storage area files to reside on another disk, you must specify the **FILENAME** argument with a full file specification.

However, you may choose to create a multfile database even if your main purpose in creating the storage area is not to distribute storage area files across more than one disk. For instance, a multfile database enables you to:

- Take advantage of hashed indexes. Hashed indexes require a storage area with mixed page format and cannot be stored in the **RDB\$SYSTEM** storage area.

CREATE STORAGE AREA Clause

- Set attributes such as page size to better correspond with tables that will be stored in the storage area.

INTERVAL IS number-data-pages

Specifies the number of data pages between SPAM pages in the storage area file, and thus the maximum number of data pages each SPAM page manages. The default, and also the minimum interval, is 216 data pages. The first page of each storage area is a SPAM page. The interval you specify determines where subsequent SPAM pages are to be inserted if there are enough data pages in the storage file to require more SPAM pages.

You cannot specify the INTERVAL storage area parameter unless you also explicitly specify PAGE FORMAT IS MIXED.

Oracle Rdb calculates the maximum INTERVAL size based on the number of blocks per page, and returns an error message if you exceed this value. For example, when the page size is 2 blocks, the maximum INTERVAL is 4008 pages. If you try to create a storage area with the INTERVAL set to 4009, Oracle Rdb returns the following error message:

```
%RDB-E-BAD_DPB_CONTENT, invalid database parameters in the database parameter
block (DPB)
-RDMS-F-SPIMAX, spam interval of 4009 is more than the Rdb maximum of 4008
-RDMS-F-AREA_NAME, area NEW
```

For more information about setting space area management parameters, see the *Oracle Rdb Guide to Database Maintenance*.

LOCKING IS ROW LEVEL

LOCKING IS PAGE LEVEL

Specifies if locking is at the page or row level for the storage area. This clause provides an alternative to requesting locks on records. Specifying a lock level when you create a storage area overrides the database default lock level. The default is ROW LEVEL.

When many records are accessed in the same area and on the same page, the LOCKING IS PAGE LEVEL clause reduces the number of lock operations performed to process a transaction; however, this is at the expense of reduced concurrency because these pages' locks are held until COMMIT/ROLLBACK. Transactions that benefit most with page-level locking are of short duration and also access several database records on the same page. However, to guarantee consistency of the data in the absence of row locking these page level locks must be held until the transaction ends with COMMIT or ROLLBACK.

Use the LOCKING IS ROW LEVEL if transactions are long in duration and lock many rows.

CREATE STORAGE AREA Clause

The LOCKING IS PAGE LEVEL clause causes fewer blocking ASTs and provides better response time and utilization of system resources. However, there is a higher contention for pages and increased potential for deadlocks and long transactions may use excessive locks.

Page-level locking is *never* applied to RDB\$SYSTEM or the DEFAULT storage-area, either implicitly or explicitly, because the locking protocol can stall metadata users.

You cannot specify page-level locking on single-file databases.

MAXIMUM OF max-pages PAGES

Specifies the maximum number of pages of each extent. The default is 9,999 pages.

MINIMUM OF min-pages PAGES

Specifies the minimum number of pages of each extent. The default is 99 pages.

NO ROW CACHE

Specifies that a row cache is not assigned to the specified storage area in the database. You cannot specify the NO ROW CACHE clause if you specify the CACHE USING clause.

Alter the storage area and name a row cache with the CACHE USING clause to assign a row cache to the storage area or to override the database default. Only one row cache is allowed for each storage area.

PAGE FORMAT IS UNIFORM

PAGE FORMAT IS MIXED

Specifies the on-disk structure for the storage area.

- The default is PAGE FORMAT IS UNIFORM. A storage area with uniform page format is a file that is divided into groups of n pages, called **clumps**, where n equals the buffer size divided by the page size. Both buffer size and page size are user specified values. By default, the buffer size is 6 blocks, and the page size is 1024 bytes or 2 blocks long, resulting in clumps of three pages. The PAGE FORMAT IS UNIFORM argument creates a storage area file that is divided into clumps. A set of clumps forms a **logical area** that can contain rows from a single table or index only.

Uniform page format storage areas generally give the best performance if the tables in the storage area are likely to be subject to a wide range of queries.

CREATE STORAGE AREA Clause

- The **PAGE FORMAT IS MIXED** argument creates a storage area with a format that allows rows from more than one table to reside on or near a particular page of the storage area file. This is useful for storing related rows from different tables on the same page of the data file. For storage areas subject to repeated queries that retrieve those related rows, a mixed page format can greatly reduce input/output overhead if the mix of rows on the page is carefully controlled. However, mixed page format storage areas degrade performance if the mix of rows on the page is not suited for the queries made against the storage area.

For more information on the relative advantages and disadvantages of uniform and mixed storage areas, see the *Oracle Rdb Guide to Database Maintenance*.

PAGE SIZE IS page-blocks BLOCKS

The size in blocks of each data page in the storage area. Page size is allocated in 512-byte blocks. The default is 2 blocks (1024 bytes). If your largest row is larger than approximately 950 bytes, allocate more blocks per page to prevent fragmented rows. If you specify a page size larger than the buffer size, an error message is returned.

PERCENT GROWTH IS growth

Specifies the percent growth of each extent. The default is 20 percent growth.

SNAPSHOT ALLOCATION IS snp-pages PAGES

Specifies the number of pages allocated for the snapshot file.

The default is 100 pages.

SNAPSHOT EXTENT IS extent-pages PAGES

SNAPSHOT EXTENT IS (extension-options)

Specifies the number of pages of each snapshot or storage area file extent. The default extent for storage area files is 100 pages.

Specify a number of pages for simple control over the extension. For greater control, and particularly for multivolume databases, use the **MIN**, **MAX**, and **PERCENT GROWTH** extension options instead.

If you use the **MIN**, **MAX**, and **PERCENT GROWTH** parameters, you must enclose them in parentheses.

SNAPSHOT FILENAME file-spec

Provides a separate file specification for the snapshot file. The **SNAPSHOT FILENAME** argument can only be specified with multifile databases.

CREATE STORAGE AREA Clause

This argument lets you specify a different file name, device, or directory for the snapshot file created by the CREATE STORAGE AREA clause. Do not specify a file extension other than .snp to the file specification. Oracle Rdb assigns the extension .snp to the file specification, even if you specify an alternate extension.

If you omit the SNAPSHOT FILENAME argument, the snapshot file gets the same device, directory, and file name as the storage area file.

STORAGE AREA area-name

Specifies the name of the storage area you want to create. The name cannot be the same as any other storage area definition in the database.

STORAGE AREA RDB\$SYSTEM

Specifies that you want the CREATE STORAGE AREA clause to override the default characteristics for the main storage area, RDB\$SYSTEM, in a new database.

The RDB\$SYSTEM storage area contains database system tables and indices. If an alternate DEFAULT STORAGE AREA is not assigned then this area may also contain unmapped user tables and indices.

THRESHOLDS ARE (val1 [,val2 [,val3]])

Specifies one, two, or three threshold values for mixed format pages. The threshold values represent a fullness percentage on a data page and establish three possible ranges of guaranteed free space on the data pages. When a data page reaches the percentage defined by a given threshold value, the space area management (SPAM) entry for the data page is updated to reflect the new fullness percentage and its remaining free space.

The default threshold values for mixed areas, if not specified, are (70,85,95), which indicates that the nominal record size should be used for SPAM threshold calculations. Oracle Rdb never stores a record on a page at the third threshold. The value you set for the highest threshold can be used to reserve space on the page for future record growth.

When only val1 is specified, this is equivalent to (val1, 100, 100). When val1 and val2 are specified, this is equivalent to (val1, val2, 100). The trailing, unspecified thresholds default to 100 percent. For example, THRESHOLDS ARE (40) would appear as (40, 100, 100).

You cannot specify the THRESHOLDS storage area parameter unless you also explicitly specify PAGE FORMAT IS MIXED.

For more information about setting space area management parameters, see the *Oracle Rdb Guide to Database Maintenance*.

CREATE STORAGE AREA Clause

Usage Notes

- You cannot use the CREATE STORAGE AREA clause with single-file databases. The presence or absence of a CREATE STORAGE AREA clause in a CREATE DATABASE statement is what determines whether the database is single file or multifile. SQL creates a multifile database only when the CREATE DATABASE statement includes at least one CREATE STORAGE AREA clause.
- The CREATE STORAGE AREA clause does not control which tables or indices will actually be associated with the storage area. The CREATE STORAGE MAP and CREATE INDEX statements control what is stored in a particular storage area file. For information about storing lists, see the CREATE STORAGE MAP Statement.
- If the LOCKING IS PAGE LEVEL or LOCKING IS ROW LEVEL clause is specified at the database level (using the ALTER DATABASE or CREATE DATABASE statements), all storage areas are affected (with the exception of RDB\$SYSTEM which is always set to row-level locking). If specified at the storage area level (using the CREATE STORAGE AREA clause), only the specified storage area attributes are affected.
- Adding a new area with a page size smaller than the smallest existing page size requires exclusive database access.

Examples

Example 1: Defining a multifile database

This example shows the definition of a database and storage areas for a multifile database.

CREATE STORAGE AREA Clause

```
SQL> -- Note that there is no semicolon before
SQL> -- the first CREATE STORAGE AREA clause.
SQL> CREATE DATABASE ALIAS MULTIFILE_EXAMPLE
cont> FILENAME 'DB_DATA01:[DB.DATA]MULTIFILE_EXAMPLE'
cont> CREATE STORAGE AREA EMPID_LOW
cont> FILENAME 'DB_DATA02:[DB.DATA]EMPID_LOW'
cont> ALLOCATION IS 10 PAGES
cont> -- Notice that the snapshot file resides on a
cont> -- different disk than the storage area file. This
cont> -- strategy reduces disk input/output bottlenecks:
cont> SNAPSHOT FILENAME 'DB_SNAP03:[DB.SNAP]EMPID_LOW'
cont> SNAPSHOT ALLOCATION IS 10 PAGES
cont> --
cont> CREATE STORAGE AREA EMPID_MID
cont> FILENAME 'DB_DATA04:EMPID_MID'
cont> ALLOCATION IS 10 PAGES
cont> SNAPSHOT FILENAME 'DB_SNAP05:[DB.SNAP]EMPID_MID'
cont> SNAPSHOT ALLOCATION IS 10 PAGES
cont> --
cont> CREATE STORAGE AREA EMPID_OVER
cont> FILENAME 'DB_DATA06:[DB.DATA]EMPID_OVER'
cont> ALLOCATION IS 10 PAGES
cont> SNAPSHOT FILENAME 'DB_SNAP07:[DB.SNAP]EMPID_OVER'
cont> SNAPSHOT ALLOCATION IS 10 PAGES
cont> --
cont> CREATE STORAGE AREA HISTORIES
cont> FILENAME 'DB_DATA02:[DB.DATA]HISTORIES'
cont> ALLOCATION IS 10 PAGES
cont> SNAPSHOT FILENAME 'DB_SNAP03:[DB.SNAP]HISTORIES'
cont> SNAPSHOT ALLOCATION IS 10 PAGES
cont> --
cont> CREATE STORAGE AREA CODES
cont> FILENAME 'DB_DATA04:[DB.DATA]CODES'
cont> ALLOCATION IS 10 PAGES
cont> SNAPSHOT FILENAME 'DB_SNAP05:[DB.SNAP]CODES'
cont> SNAPSHOT ALLOCATION IS 10 PAGES
cont> --
cont> CREATE STORAGE AREA EMP_INFO
cont> FILENAME 'DB_DATA08:[DB.DATA]EMP_INFO'
cont> ALLOCATION IS 10 PAGES
cont> SNAPSHOT FILENAME 'DB_SNAP09:[DB.SNAP]EMP_INFO'
cont> SNAPSHOT ALLOCATION IS 10 PAGES
cont> --
cont> -- End the CREATE DATABASE statement:
cont> ;
```

CREATE STORAGE AREA Clause

Example 2:

This example shows how to set page-level and row-level locking on storage areas from both the database level and from the storage area level.

```
SQL> CREATE DATABASE FILENAME sample
cont>   LOCKING IS PAGE LEVEL
cont> --
cont> -- All storage areas will default to page-level locking unless
cont> -- explicitly set to row-level locking.
cont> --
cont> CREATE STORAGE AREA RDB$SYSTEM
cont>   FILENAME sample_system
cont> --
cont> -- You cannot specify page-level locking on RDB$SYSTEM.  RDB$SYSTEM
cont> -- always defaults to row-level locking.
cont> --
cont> CREATE STORAGE AREA HASH_AREA
cont>   FILENAME sample_hash
cont>   PAGE FORMAT IS MIXED
cont> --
cont> -- HASH_AREA defaults to page-level locking.
cont> --
cont> CREATE STORAGE AREA DATA_AREA
cont>   FILENAME sample_data
cont>   LOCKING IS ROW LEVEL
cont> --
cont> -- DATA_AREA is explicitly set to row-level locking.
cont> --
cont> ;
SQL> SHOW STORAGE AREAS (ATTRIBUTES) *
Storage Areas in database with filename sample

RDB$SYSTEM
List storage area.
Access is:      Read write
Page Format:    Uniform
Page Size:     2 blocks
.
.
.
Extent :       Enabled
Locking is Row Level
```

CREATE STORAGE AREA Clause

```
HASH_AREA
  Access is:      Read write
  Page Format:    Mixed
  Page Size:     2 blocks
  .
  .
  .
  Extent :       Enabled
  Locking is Page Level

DATA_AREA
  Access is:      Read write
  Page Format:    Uniform
  Page Size:     2 blocks
  .
  .
  .
  Extent :       Enabled
  Locking is Row Level
```

See the SHOW Statement for information on the SHOW STORAGE AREAS statement.

Example 3: Creating and assigning a row cache to a storage area

```
SQL> create database
cont>   filename SAMPLE_DB
cont>   reserve 2 cache slots
cont>   row cache is enabled
cont>   default storage area is AREA1
cont>   create cache CACHE1
cont>   cache size is 1000 rows
cont>   row length is 1000 bytes
cont>   create storage area AREA1
cont>   cache using CACHE1
cont> ;
SQL> show cache CACHE1

CACHE1
  Cache Size:      1000 rows
  Row Length:     1000 bytes
  Row Replacement: Enabled
  Shared Memory:  Process
  Large Memory:   Disabled
  Window Count:  100
  Working Set Count: 10
  Reserved Rows: 20
  Allocation:    100 blocks
  Extent:        100 blocks
SQL> show storage area AREA1
```

CREATE STORAGE AREA Clause

```
AREA1
  Access is:      Read write
  Page Format:    Uniform
  Page Size:     2 blocks
  Area File:     USER_DISK:[DOC.DATABASES]AREA1.RDA;1
  Area Allocation: 702 pages
  Extent:        Enabled
  Area Extent Minimum: 99 pages
  Area Extent Maximum: 9999 pages
  Area Extent Percent: 20 percent
  Snapshot File: USER_DISK:[DOC.DATABASES]AREA1.SNP;1
  Snapshot Allocation: 100 pages
  Snapshot Extent Minimum: 99 pages
  Snapshot Extent Maximum: 9999 pages
  Snapshot Extent Percent: 20 percent
Locking is Row Level
  Using Cache CACHE1

Database objects using Storage Area AREA1:
Usage      Object Name      Map / Partition
-----
Default Area
```

CREATE STORAGE MAP Statement

CREATE STORAGE MAP Statement

Associates a table with one or more storage areas in a multifile database. The CREATE STORAGE MAP statement specifies a **storage map** that controls which lists or rows of a table are stored in which storage areas.

In addition to creating storage maps, the CREATE STORAGE MAP statement has options that control:

- Which index the database system uses when inserting rows in the table
- Whether or not the rows of the table are stored in a compressed format
- Whether or not partitioning keys can be modified.
- Whether the table is partitioned vertically, horizontally, or both.
- Whether logging is enabled or disabled for the duration of this operation

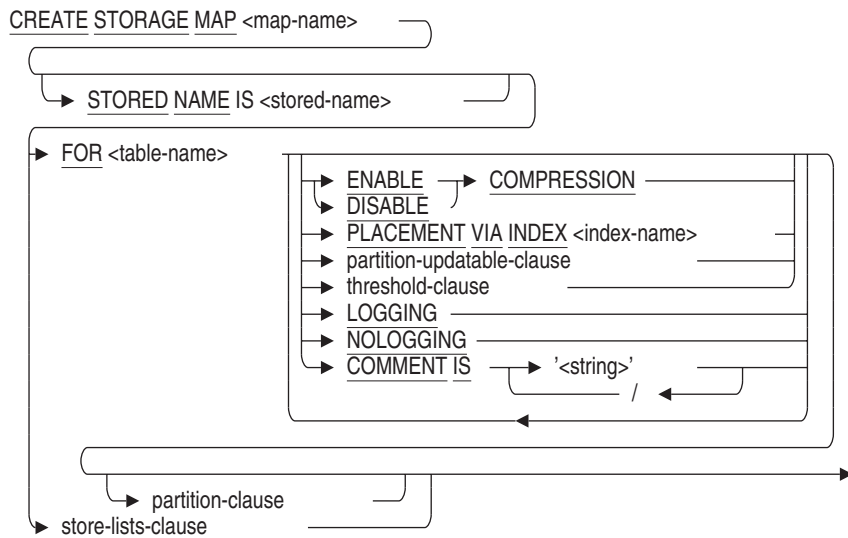
Environment

You can use the CREATE STORAGE MAP statement:

- In interactive SQL
- Embedded in host language programs to be precompiled
- As part of a procedure in an SQL module
- In dynamic SQL as a statement to be dynamically executed

Format

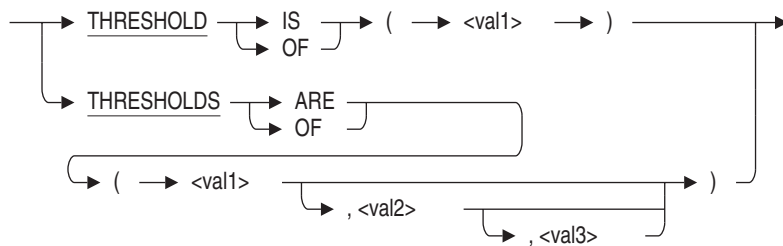
CREATE STORAGE MAP Statement



partition-updatable-clause =



threshold-clause =



partition-clause =

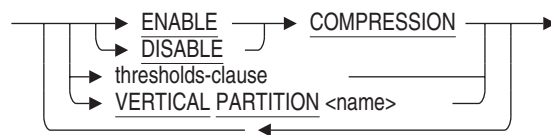


CREATE STORAGE MAP Statement

columns-clause =



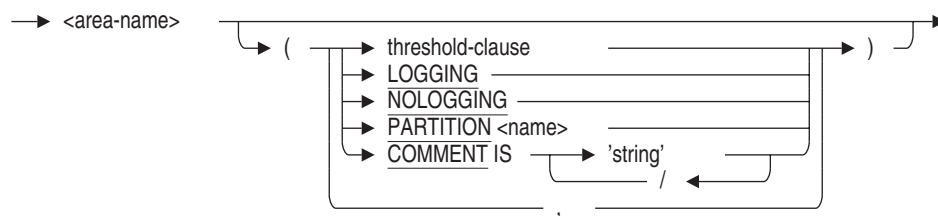
store-attributes =



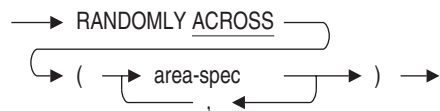
store-clause =



area-spec =

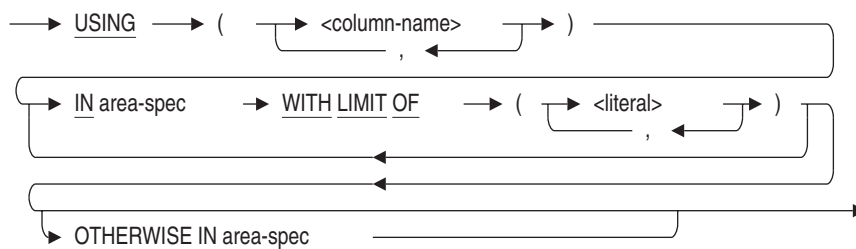


across-clause =

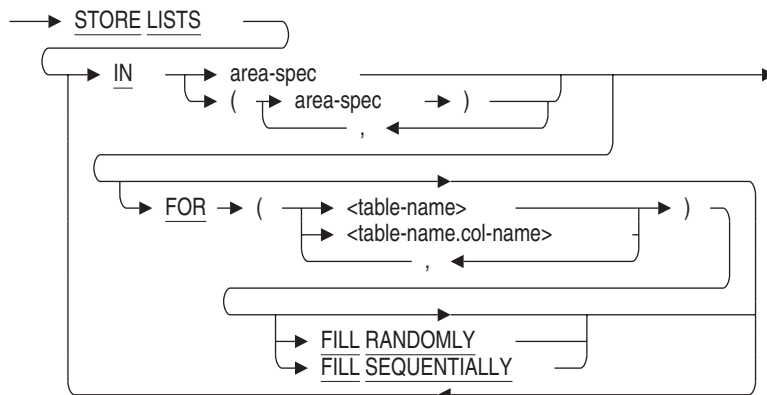


CREATE STORAGE MAP Statement

using-clause =



store-lists-clause =



Arguments

across-clause

Associates the table with two or more storage areas.

COMMENT IS 'string'

Adds a comment about the storage map. SQL displays the text of the comment when it executes a `SHOW STORAGE MAPS` statement. Enclose the comment in single quotation marks (`'`) and separate multiple lines in a comment with a slash mark (`/`).

ENABLE COMPRESSION

DISABLE COMPRESSION

Specifies whether the rows for the partition are compressed or uncompressed when stored. You can enable or disable compression on each vertical partition.

CREATE STORAGE MAP Statement

You enable compression to conserve disk space, but there is a small CPU overhead for inserting and retrieving compressed rows.

If you omit this clause, the default compression is that which was specified for the storage map before the first STORE COLUMNS clause. The default is ENABLE COMPRESSION.

FILL RANDOMLY **FILL SEQUENTIALLY**

Specifies whether to fill the area set randomly or sequentially. Specifying FILL RANDOMLY or FILL SEQUENTIALLY requires a FOR clause. When a storage area is filled, it is removed from the list of available areas. Oracle Rdb does not attempt to store any more lists in that area during the current database attach. Instead, Oracle Rdb starts filling the next specified area.

When a set of areas is filled sequentially, Oracle Rdb stores lists in the first specified area until that area is filled.

If the set of areas is filled randomly, lists are stored across multiple areas. This is the default. Random filling is intended for read/write media, which will benefit from the I/O distribution across the storage areas.

The keywords FILL RANDOMLY and FILL SEQUENTIALLY can only be applied to areas contained within an area list.

FOR (table-name)

Specifies the table or tables to which this list storage map applies. The named table must already be defined. If you want to store lists of more than one table in the storage area, separate the names of the tables with commas. For each area, you can specify one FOR clause and list of table names.

FOR (table-name.col-name)

Specifies the name of the table and column containing the list to which this storage map applies. Separate the table name and the column name with a period (.). The named table and column must already be defined. If you want to store multiple lists in the storage area, separate the table name and column name combinations with commas. For each area, you can specify one FOR clause and a list of column names.

LOGGING **NOLOGGING**

The LOGGING clause specifies that the CREATE STORAGE MAP statement should be logged in the recovery-unit journal file (.ruj) and after-image journal file (.ajj).

CREATE STORAGE MAP Statement

The NOLOGGING clause specifies that the CREATE STORAGE MAP statement should not be logged in the recovery-unit journal file (.ruj) and after-image journal file (.aij).

The LOGGING clause is the default.

OTHERWISE IN area-name

For partitioned storage maps only, specifies the storage area that is used as the overflow partition. An **overflow partition** is a storage area that holds any values that are higher than those specified in the WITH LIMIT OF clause. An overflow partition holds those values that exceed the highest specified limits.

partition-clause

Defines vertical partitioning, horizontal partitioning, or both for the specified table.

Horizontal partitioning means that you divide the rows of the table among storage areas according to data values in one or more columns. **Vertical partitioning** means that you divide the columns of the table among storage areas. A given storage area will then contain only some of the columns of a table. You can combine both horizontal and vertical partitions in a single map.

Vertical partitioning reduces disk I/O operations by placing frequently used data in one area, so that you can read and update those portions of the table in a single disk I/O operation.

See the *Oracle Rdb Guide to Database Design and Definition* for more information regarding partitioning.

PARTITION name

Names the partition. The name can be a delimited identifier if the dialect or quoting rules are set to SQL92 or SQL99. Partition names must be unique within the storage map. If you do not specify this clause, Oracle Rdb generates a default name for the partition.

PARTITIONING IS NOT UPDATABLE

Specifies that the value of the partitioning key cannot be modified and that the row is always stored in the storage area based on the partitioning criteria in the STORE USING clause. The partitioning key is the column or list of columns specified in the STORE USING clause.

Specifying the PARTITIONING IS NOT UPDATABLE clause allows Oracle Rdb to quickly retrieve data because the partitioning criteria can be used when optimizing the query.

CREATE STORAGE MAP Statement

To update columns that are partitioning keys in a NOT UPDATABLE storage map, you must delete the rows and then reinsert the rows to ensure that they are placed in the correct location.

If you specify the PARTITIONING clause, you must also specify the STORE USING clause when defining a storage map.

If the PARTITIONING clause is not specified, UPDATABLE is the default.

See the *Oracle Rdb Guide to Database Design and Definition* for more information regarding partitioning.

PARTITIONING IS UPDATABLE

Specifies that the partitioning key can be modified. The partitioning key is the column or list of columns specified in the STORE USING clause.

If you modify a row in an UPDATABLE storage map, the row is not moved to a different storage area even if the new value of the partitioning key is not within the limits of original storage area. As a result, Oracle Rdb must consider all storage areas specified in the STORE USING clause when retrieving a row.

If you specify the PARTITIONING clause, you must also specify the STORE USING clause when defining a storage map.

If the PARTITIONING clause is not specified, UPDATABLE is the default.

See the *Oracle Rdb Guide to Database Design and Definition* for more information regarding partitioning.

PLACEMENT VIA INDEX index-name

Directs the database system to store a column in a way that optimizes access to that column by the indicated path. Oracle Rdb chooses a target page for any columns being stored by rules that take into account the type of index named (sorted or hashed), the type of storage areas involved (uniform or mixed), and how indexes and tables are assigned to storage areas.

For a hashed index, Oracle Rdb calculates the page containing the hashed index node that points to the column. If that page is within the same storage area in which the column will be stored, it is used as the target page for storing the column. If that page is not within the same storage area in which the column is to be stored, Oracle Rdb chooses a target page in the same relative position within the appropriate storage area (if it is a mixed storage area) or a page in a clump reserved for that table (if it is a uniform storage area).

CREATE STORAGE MAP Statement

For a sorted index, Oracle Rdb finds the database key of the next lowest row to the one being stored and uses the page number in the database key as the target page.

STORAGE MAP map-name

Specifies the name of the storage map you want to create. The name cannot be the same as any other definition in the database.

store-clause

The storage map definition. The store-clause in a CREATE STORAGE MAP statement lets you specify which storage area files are used to store rows from the table.

- All rows of a table can be associated with a single storage area.
- Rows of a table can be randomly distributed among several storage areas.
- Rows of a table can be systematically distributed, or partitioned, among several storage areas by specifying upper limits on the values for a column in a particular storage area. This is called horizontal partitioning.
- Columns of a table can be partitioned among storage areas. This is called vertical partitioning.

If you omit the storage map definition, the default is to store all the rows for a table in the default storage area. See the CREATE and IMPORT DATABASE statements for information on the default storage area.

STORE COLUMNS (column-name)

Lists the columns which will be stored in the subsequent map.

Multiple STORE COLUMNS clauses may appear in a map to spread across multiple storage areas. A column name may only appear in one STORE COLUMNS clause. A final STORE clause can appear to provide a location for all remaining unspecified columns.

STORE IN area-name

Associates the table directly with a single storage area. All rows in the table are stored in the area you specify.

STORE LISTS IN area-name

Directs the database system to store the lists from tables in a specified storage area or in a set of areas. You can create only one storage map for lists within each database.

CREATE STORAGE MAP Statement

You must specify the default storage area for lists in the `STORE LISTS` clause. The default list storage area contains lists from system tables as well as lists not directed elsewhere by the `STORE LISTS` clause. You can also use the `LIST STORAGE AREA` clause of the `CREATE DATABASE` statement to specify a default storage area for lists. If you do not use the `STORE LISTS` clause and do not specify a list storage area in the `CREATE DATABASE` statement, Oracle Rdb uses the default storage area as the default list storage area. The following example directs Oracle Rdb to place all lists in the `LISTS` storage area unless otherwise specified in a storage map:

```
SQL> CREATE DATABASE FILENAME mf_personnel
SQL> LIST STORAGE AREA IS LISTS
SQL> CREATE STORAGE AREA LISTS;
```

The accompanying storage map statement must also specify the `LISTS` storage area as the default storage area.

```
SQL> CREATE STORAGE MAP LISTS_MAP
cont> STORE LISTS IN LISTS1 FOR (EMPLOYEES.RESUME)
cont>          IN LISTS;
```

You can use an area set to specify that data is to be distributed across several areas. The following example shows how you can store data in three storage areas (`LISTS1`, `LISTS2`, and `LISTS3`) for two different columns in `TABLE1`. The default list storage area is `LISTS1`.

```
CREATE STORAGE MAP LISTS_MAP
      STORE LISTS IN (LISTS1,LISTS2,LISTS3) FOR (TABLE1.COL1, TABLE1.COL2)
      IN LISTS1;
```

You can store lists from different tables in the same area. The following example shows how you can store data from `TABLE1`, `TABLE2`, and `TABLE3` in the `LISTS` storage area. The default list storage area is `RDB$SYSTEM`.

```
SQL> CREATE STORAGE MAP LISTS_MAP -- to direct the list data to area LISTS
cont> STORE LISTS IN LISTS FOR (TABLE1, TABLE2, TABLE3)
cont>          IN RDB$SYSTEM;
```

Alternatively, you can store lists from each table in unique areas. The following example shows list data from `TABLE1` being stored in the `LISTS1` storage area and list data from `TABLE2` being stored in the `LISTS2` storage area. The default list storage area is `RDB$SYSTEM`.

```
CREATE STORAGE MAP LISTS_MAP
      STORE LISTS IN LIST1 FOR (TABLE1)
      IN LIST2 FOR (TABLE2)
      IN RDB$SYSTEM;
```

CREATE STORAGE MAP Statement

You can also specify that different columns from the same table go into different areas. The following example shows data from different columns in TABLE1 being stored in either LISTS1 or LISTS2. The default list storage area is RDB\$SYSTEM.

```
CREATE STORAGE MAP LISTS_MAP
  STORE LISTS IN LISTS1 FOR (TABLE1.COL1)
              IN LISTS2 FOR (TABLE1.COL2)
              IN RDB$SYSTEM;
```

STORE RANDOMLY ACROSS (area-name)

As rows are inserted in the table, they are distributed randomly across the storage areas named in the list. You must name at least two storage areas in this clause.

STORE USING (column-name) IN area-name

The database system compares values in the columns to the values in the WITH LIMIT OF clause to determine placement of rows inserted into the table. For instance, a storage map with the clause STORE USING (X,Y,Z) IN AREA1 WITH LIMIT OF (1,2,3) means that a row must meet these criteria to be stored in AREA1:

$$(X < 1) \text{ OR } ((X = 1) \text{ AND } ((Y < 2) \text{ OR } ((Y = 2) \text{ AND } (Z \leq 3))))$$

Use RMU EXTRACT to have the store using expression expanded. See Example 9.

STORED NAME IS stored-name

Specifies a name that Oracle Rdb uses to access a storage map created in a multiscHEMA database. The stored name allows you to access multiscHEMA definitions using interfaces that do not recognize multiple schemas in one database. You cannot specify a stored name for a storage map in a database that does not allow multiple schemas. For more information on stored names, see Section 2.2.18.

threshold-clause

Specifies one, two, or three default threshold values for logical areas in storage areas with uniform format pages. The threshold values (val1, val2, and val3) represent a fullness percentage on a data page and establish three possible ranges of guaranteed free space on the data pages. When a data page reaches the percentage defined by a given threshold value, the space area management (SPAM) entry for the data page is updated to reflect the new fullness percentage and its remaining free space.

CREATE STORAGE MAP Statement

Oracle Rdb never stores a record at the third threshold. The value you set for the highest threshold can be used to reserve space on the page for future record growth.

When only val1 is specified, this is equivalent to (val1, 100, 100). When val1 and val2 are specified, this is equivalent to (val1, val2, 100). The trailing, unspecified thresholds default to 100 percent. For example, THRESHOLDS ARE (40) would appear as (40, 100, 100).

If no thresholds are specified for the area, the default is (0,0,0). This causes the SPAM algorithm to set thresholds based on the nominal record length for the logical area; for example, the node size for the index or the uncompressed length of the row for a table.

You cannot specify the thresholds for the storage map attribute for any area that is a mixed page format. If you have a mixed page format, set the thresholds for the storage area using the ADD STORAGE AREA or CREATE STORAGE AREA clause of the ALTER DATABASE, CREATE DATABASE, or IMPORT statements.

VERTICAL PARTITION name

Names a vertical partition. The name can be a delimited identifier if the dialect or quoting rules are set to SQL92 or SQL99. Partition names must be unique within the storage map. If you do not specify this clause, Oracle Rdb generates a default name for the partition.

using-clause

Specifies columns whose values are used as limits for partitioning the table horizontally across multiple storage areas.

WITH LIMIT OF (literal)

Specifies the maximum values that the columns named in the USING clause can have when rows are initially stored in the specified storage area. Repeat this clause to partition the rows of a table among multiple storage areas.

The number of literals listed must be the same as the number of columns in the USING clause. The data type of the literals must agree with the data type of the column. For character columns, enclose the literals in single quotation marks.

The values in the WITH LIMIT OF clause only affect placement of rows when they are initially stored. If UPDATE statements change data in a row so that values in columns named in the USING clause exceed values specified in the WITH LIMIT OF clause, the row is not moved into a different storage area.

CREATE STORAGE MAP Statement

Usage Notes

- The CREATE STORAGE MAP statement creates a SQL mapping routine that matches the WITH LIMIT OF clause for the storage map. The routine is automatically created in the system module RDB\$STORAGE_MAPS (use SHOW SYSTEM MODULES to view). The storage map name is used to name the mapping routine (use SHOW SYSTEM FUNCTIONS to view).

Note

If a routine already exists with the same name as the storage map, then the mapping routine will not be created.

If the storage map includes a STORE COLUMNS clause, that is, a vertically partitioned map, then several routines will be created and uniquely named by adding the vertical partition number as a suffix.

The mapping routine returns the following values:

- Zero (0) if the storage map is defined as RANDOMLY ACROSS. This routine is just a descriptive place holder.
- Positive value representing the storage map number (the same value as stored in RDB\$ORDINAL_POSITION column of the RDB\$STORAGE_MAP_AREAS table). These values can be used with the PARTITION clause of the SET TRANSACTION...RESERVING clause to reserve a specific partition prior to inserting the row.
- A value of -1 if the storage map has no OTHERWISE clause. This indicates that the row cannot be inserted because it does not match any of the WITH LIMIT OF clauses.
- You must specify either a STORE clause, a PLACEMENT clause, or a COMPRESSION clause in a CREATE STORAGE MAP statement.
- If you specify multiple storage areas in a CREATE STORAGE MAP statement, they must have the same format; you cannot specify both MIXED and UNIFORM format storage areas in the same storage map.
- You cannot create more than one map for the rows from a given table, but you can create one map for that table's rows and a separate map for that table's lists.

CREATE STORAGE MAP Statement

- If you repeat a column or table in the storage map with a different area, then all columns of data type LIST OF BYTE VARYING are stored randomly across the specified areas, unless you specify SEQUENTIAL storage.
- You cannot delete a list storage map from the database.
- You can only specify one PLACEMENT VIA INDEX clause per storage map.
- Attempts to create a storage map fail if that storage map or its affected table is involved in a query at the same time. Users must detach from the database with a DISCONNECT statement before you can create the storage map. When Oracle Rdb first accesses an object such as the table, a lock is placed on that object and not released until the user exits the database. If you attempt to update this object, you get a LOCK CONFLICT ON CLIENT message due to the other user's access to the object.
- You cannot execute the CREATE STORAGE MAP statement when the RDB\$SYSTEM storage area is set to read-only. You must first set RDB\$SYSTEM to read/write. See the *Oracle Rdb7 Guide to Database Performance and Tuning* for more information on the RDB\$SYSTEM storage area.
- If a storage map does not contain an overflow partition (defined by the OTHERWISE clause), you can add new partitions to the storage map without reorganizing the storage areas. For example:

```
SQL> ALTER STORAGE MAP EMP_MAP
cont>   STORE USING (EMPLOYEE_ID)
cont>     IN PERSONNEL_1 WITH LIMIT OF ('00399')
cont>     IN PERSONNEL_2 WITH LIMIT OF ('00699')
cont>     IN PERSONNEL_3 WITH LIMIT OF ('10000')
cont>     IN PERSONNEL_4 WITH LIMIT OF ('10399');
SQL>
```

Because the original storage map did not contain an OTHERWISE clause, you do not need to reorganize the storage areas.

For more information, see the *Oracle Rdb Guide to Database Design and Definition* and the *Oracle Rdb7 Guide to Database Performance and Tuning*.

- If you attempt to insert values that are out of range of the storage map, you receive an error similar to the following:

```
%RDMS-E-EXCMAPLIMIT, exceeded limit on last partition in storage map for
EMPLOYEES
```

Your applications should include code that handles this type of error.

CREATE STORAGE MAP Statement

- If a storage map contains an overflow partition and you want to alter the storage map to change the overflow partition to a partition defined with the WITH LIMIT OF clause, you must use the REORGANIZE clause if you want existing data that is stored in the overflow partition moved to appropriate storage area. For example:

```
SQL> ALTER STORAGE MAP JH_MAP
cont>   STORE USING (EMPLOYEE_ID)
cont>     IN PERSONNEL_1 WITH LIMIT OF ('00399')
cont>     IN PERSONNEL_2 WITH LIMIT OF ('00699')
cont>     IN PERSONNEL_3 WITH LIMIT OF ('10000')
cont>     IN PERSONNEL_4 WITH LIMIT OF ('10399')
cont>   REORGANIZE;
SQL>
```

- Oracle Rdb checks to ensure that list maps are not created on system tables.
- You can create a storage map for an existing table that contains data. However, the following restrictions apply:
 - The storage map must be a simple map that references only the default storage area and represents the default mapping for the table.
 - You cannot change the thresholds or compression for the table.
 - You cannot specify the PLACEMENT VIA INDEX clause.
 - The storage map cannot be vertically partitioned.
 - The storage map may not include a WITH LIMIT clause for the storage area.

Once the storage map is created, you can use the ALTER STORAGE MAP statement to reorganize the table as needed. This is shown in Example 6.

If the new storage map contains any unacceptable attributes it will be rejected, as shown in Example 7 in the Examples section.

- You must specify the columns-clause to vertically partition a storage map.
- You cannot alter a vertically partitioned storage map once it is defined.
- Columns not specified in the columns-clause are mapped to the final vertical partition.
- The final vertical partition holds all unmapped columns and is used by future ALTER TABLE . . . ADD COLUMN statements. Only the final STORE clause can omit the COLUMNS clause.
- If you are not vertically partitioning a storage map, only one store-clause is allowed in the storage map definition.

CREATE STORAGE MAP Statement

- Some system tables are automatically created in the secondary system area if defined by the clause `DEFAULT STORAGE AREA` in the `CREATE DATABASE` statement. Additionally, a set of optional system tables exists (which may not exist in all databases) that can be mapped manually to other storage areas.

The set of system tables for which you can change the mapping, and the instructions on how to do so, are provided in the section on moving certain system tables to separate storage areas in the *Oracle Rdb Guide to Database Design and Definition*.

Examples

Example 1: Defining storage maps for a multifile database

This example shows the definition of storage maps for a multifile database. The tables named in the `CREATE STORAGE MAP` statements have the same definitions as those in the sample database. See the `CREATE STORAGE AREA Clause` for an example of a `CREATE DATABASE` statement with `CREATE STORAGE AREA` clauses that create the storage areas referred to in this example.

```
SQL> -- Declare the database as the default:
SQL> ATTACH 'FILENAME multifile_example';
SQL> --
SQL> CREATE STORAGE MAP EMPLOYEE_MAP FOR EMPLOYEEES
cont> STORE USING (EMPLOYEE_ID)
cont>   IN EMPID_LOW WITH LIMIT OF ('00200')
cont>   IN EMPID_MID WITH LIMIT OF ('00500')
cont>   OTHERWISE IN EMPID_OVER;
SQL> --
SQL> CREATE STORAGE MAP RESUME_MAP
cont> STORE LISTS IN EMP_INFO FOR (TABLE1, TABLE2, TABLE3)
cont>   IN RDB$SYSTEM;
SQL> --
SQL> CREATE STORAGE MAP JOB_HISTORY_MAP FOR JOB_HISTORY
cont> STORE IN HISTORIES;
SQL> --
SQL> CREATE STORAGE MAP SALARY_HISTORY_MAP FOR SALARY_HISTORY
cont> STORE IN HISTORIES;
SQL> --
SQL> CREATE STORAGE MAP JOBS_MAP FOR JOBS
cont> STORE IN CODES;
SQL> --
SQL> CREATE STORAGE MAP DEPARTMENTS_MAP FOR DEPARTMENTS
cont> STORE IN CODES;
SQL> --
SQL> CREATE STORAGE MAP COLLEGES_MAP FOR COLLEGES
cont> STORE IN CODES;
```

CREATE STORAGE MAP Statement

```
SQL> --
SQL> CREATE STORAGE MAP DEGREES_MAP FOR DEGREES
cont> STORE IN EMP_INFO;
SQL> --
SQL> CREATE STORAGE MAP WORK_STATUS_MAP FOR WORK_STATUS
cont> STORE IN HISTORIES;
SQL> --
SQL> --
SQL> -COMMIT;
SQL> --
```

CREATE STORAGE MAP Statement

Example 2: Defining storage maps that place and override thresholds on uniform storage areas

```
SQL> CREATE DATABASE FILENAME birdlist
cont> CREATE STORAGE AREA AREA1
cont> CREATE STORAGE AREA AREA2
cont> CREATE STORAGE AREA AREA3
cont> CREATE STORAGE AREA AREA4
cont> CREATE TABLE SPECIES
cont>     ( GENUS          CHAR (30),
cont>       SPECIES        CHAR (30),
cont>       COMMON_NAME    CHAR (40),
cont>       FAMILY_NUMBER  INT (3),
cont>       SPECIES_NUMBER INT (3)
cont>     )
cont> CREATE INDEX I1 ON SPECIES (FAMILY_NUMBER)
cont> CREATE TABLE SIGHTING
cont>     ( SPECIES_NUMBER INT (3),
cont>       COMMON_NAME    CHAR (40),
cont>       CITY           CHAR (20),
cont>       STATE          CHAR (20),
cont>       SIGHTING_DATE  DATE ANSI,
cont>       NOTES_NUMBER   INT (5))
cont> CREATE INDEX I2 ON SIGHTING (SPECIES_NUMBER)
cont> CREATE TABLE FIELD_NOTES
cont>     ( WEATHER        CHAR (30),
cont>       TIDE           CHAR (15),
cont>       SPECIES_NUMBER INT (3),
cont>       SIGHTING_TIME  TIMESTAMP(2),
cont>       NOTES          CHAR (500),
cont>       NOTES_NUMBER   INT (5))
cont> CREATE INDEX I3 ON FIELD_NOTES (NOTES_NUMBER)
cont> ;
SQL> --
SQL> -- The following CREATE STORAGE MAP statements place and
SQL> -- override thresholds on uniform storage area.
SQL> --
SQL> -- Note that the default threshold clause for the
SQL> -- storage map is not enclosed in parentheses, but each
SQL> -- threshold clause associated with a particular area is.
SQL> --
SQL> CREATE STORAGE MAP M1 FOR SPECIES
cont>     THRESHOLDS ARE (30, 50, 80)
cont>     ENABLE COMPRESSION
cont>     PLACEMENT VIA INDEX I1
cont>     STORE
cont>     IN AREA1
cont>     (THRESHOLD (10) );
SQL> --
SQL> CREATE STORAGE MAP M2 FOR SIGHTING
cont>     THRESHOLD IS (40)
```

CREATE STORAGE MAP Statement

```
cont>     STORE
cont>     RANDOMLY ACROSS (
cont>         AREA1 (THRESHOLD OF (10) ),
cont>         AREA2 (THRESHOLDS ARE (30, 50, 98) ),
cont>         AREA3
cont>     );
SQL> --
SQL> CREATE STORAGE MAP M3 FOR FIELD_NOTES
cont>     THRESHOLDS OF (50,70,90)
cont>     STORE
cont>         USING (SPECIES_NUMBER, NOTES_NUMBER)
cont>             IN AREA1
cont>                 (THRESHOLDS OF (20, 80, 90) )
cont>                 WITH LIMIT OF (30, 88)
cont>             IN AREA2
cont>                 WITH LIMIT OF (40, 89)
cont>             IN AREA3
cont>                 WITH LIMIT OF (50, 90)
cont>             OTHERWISE IN AREA4
cont>                 (THRESHOLDS ARE (20, 30, 40));
SQL> --
SQL> SHOW STORAGE MAP *;
User Storage Maps in database with filename birdlist
M1
For Table: SPECIES
Placement Via Index: I1
Partitioning is: UPDATABLE
Store clause: STORE
           IN AREA1
           (THRESHOLD (10) )

Partition information for storage map:
Compression is: ENABLED
Partition: (1) SYS_P00062
Storage Area: AREA1

M2
For Table: SIGHTING
Partitioning is: UPDATABLE
Store clause: STORE
RANDOMLY ACROSS (
           AREA1 (THRESHOLD OF (10) ),
           AREA2 (THRESHOLDS ARE (30, 50, 98) ),
           AREA3
)
)
```

CREATE STORAGE MAP Statement

```
Partition information for storage map:
Compression is: ENABLED
Partition: (1) SYS_P00063
Storage Area: AREA1
Partition: (2) SYS_P00064
Storage Area: AREA2
Partition: (3) SYS_P00065
Storage Area: AREA3

M3
For Table: FIELD_NOTES
Partitioning is: UPDATABLE
Store clause: STORE
    USING (SPECIES_NUMBER, NOTES_NUMBER)
        IN AREA1
            (THRESHOLDS OF (20, 80, 90) )
            WITH LIMIT OF (30, 88)
        IN AREA2
            WITH LIMIT OF (40, 89)
        IN AREA3
            WITH LIMIT OF (50, 90)
        OTHERWISE IN AREA4
            (THRESHOLDS ARE (20, 30, 40))
```

```
Partition information for storage map:
Compression is: ENABLED
Partition: (1) SYS_P00066
Storage Area: AREA1
Partition: (2) SYS_P00067
Storage Area: AREA2
Partition: (3) SYS_P00068
Storage Area: AREA3
Partition: (4) SYS_P00069
Storage Area: AREA4
```

```
SQL> --
SQL> ROLLBACK;
```

Example 3: Creating a storage map that stores lists

This example creates a storage map that stores lists on specific storage areas.

CREATE STORAGE MAP Statement

```
SQL> CREATE DATABASE FILENAME test
cont> CREATE STORAGE AREA LISTS1 PAGE FORMAT IS MIXED
cont> CREATE STORAGE AREA LISTS2 PAGE FORMAT IS MIXED
cont>
cont> CREATE TABLE EMPLOYEES
cont>     (EMP_ID CHAR(5),
cont>     RESUME LIST OF BYTE VARYING);
SQL> --
SQL> CREATE STORAGE MAP LISTS_MAP
cont>     STORE LISTS IN
cont>     (LISTS1,LISTS2) FOR (EMPLOYEES.RESUME)
cont>     FILL SEQUENTIALLY
cont>     IN RDB$SYSTEM;
```

Example 4: Creating an alternate map

This example following storage map shows an alternate mapping for the EMPLOYEES table in the same MF_PERSONNEL database.

```
SQL> create storage map EMPLOYEES_MAP
cont>     for EMPLOYEES
cont>     placement via index EMPLOYEES_HASH
cont>     -- store the primary information horizontally partitioned
cont>     -- across the areas EMPIDS_LOW, EMPIDS_MID and EMPIDS_OVER
cont>     -- disable compress because these columns are accessed often
cont>     store
cont>         columns (EMPLOYEE_ID, LAST_NAME,
cont>                 FIRST_NAME, MIDDLE_INITIAL)
cont>         disable compression
cont>         using (EMPLOYEE_ID)
cont>             in EMPIDS_LOW
cont>                 with limit of ('00200')
cont>             in EMPIDS_MID
cont>                 with limit of ('00400')
cont>             otherwise in EMPIDS_OVER
cont>
cont>     -- place all the address information in EMP_INFO
cont>     -- make sure these character columns are compressed
cont>     -- to remove the trailing spaces
cont>     store
cont>         columns (ADDRESS_DATA_1, ADDRESS_DATA_2, CITY, STATE,
cont>                 POSTAL_CODE)
cont>         enable compression
cont>         in EMP_INFO
cont>
cont>     -- the remaining columns get
cont>     -- written randomly over these areas
cont>     store
cont>         enable compression
cont>         randomly across (SALARY_HISTORY, JOBS);
```


CREATE STORAGE MAP Statement

Example 5: Disabling logging and naming horizontal and vertical partitions

```
SQL> CREATE DATABASE FILENAME birdlist
cont> CREATE STORAGE AREA AREA1
cont> CREATE STORAGE AREA AREA2
cont> CREATE STORAGE AREA AREA3
cont> CREATE STORAGE AREA AREA4
cont> CREATE STORAGE AREA AREA5
cont> CREATE STORAGE AREA AREA6
cont> CREATE STORAGE AREA AREA7
cont> CREATE STORAGE AREA AREA8
cont> CREATE TABLE SPECIES
cont> ( GENUS          CHAR (30),
cont>   SPECIES          CHAR (30),
cont>   COMMON_NAME     CHAR (40),
cont>   FAMILY_NUMBER   INT (3),
cont>   SPECIES_NUMBER  INT (3)
cont> )
cont> CREATE INDEX I1 ON SPECIES (FAMILY_NUMBER)
cont> CREATE TABLE SIGHTING
cont> ( SPECIES_NUMBER INT (3),
cont>   COMMON_NAME   CHAR (40),
cont>   CITY CHAR (20),
cont>   STATE CHAR (20),
cont>   SIGHTING_DATE DATE ANSI,
cont>   NOTES_NUMBER INT (5))
cont> CREATE INDEX I2 ON SIGHTING (SPECIES_NUMBER)
cont> CREATE TABLE FIELD_NOTES
cont> ( WEATHER CHAR (30),
cont>   TIDE CHAR (15),
cont>   SIGHTING_TIME TIMESTAMP(2),
cont>   NOTES CHAR (500),
cont>   NOTES_NUMBER INT (5),
cont>   SPECIES_NUMBER INT (3))
cont> CREATE INDEX I3 ON FIELD_NOTES (NOTES_NUMBER);
SQL> --
SQL> -- Note that the default threshold clause for the
SQL> -- storage map is not enclosed in parentheses, but each
SQL> -- threshold clause associated with a particular area is enclosed
SQL> -- in parentheses.
SQL> --
SQL> CREATE STORAGE MAP M1 FOR SPECIES
cont> THRESHOLDS ARE (30, 50, 80)
cont> ENABLE COMPRESSION
cont> PLACEMENT VIA INDEX I1
cont> NOLOGGING
cont> COMMENT IS 'Storage Map for Species'
cont> STORE
cont> IN AREA1
cont> (THRESHOLD (10),
cont>   PARTITION AREA1,
cont>   COMMENT IS 'Partition is AREA1');
```

CREATE STORAGE MAP Statement

```
SQL> --
SQL> CREATE STORAGE MAP M2 FOR SIGHTING
cont>   THRESHOLD IS (40)
cont>   STORE
cont>   RANDOMLY ACROSS (
cont>     AREA1 (THRESHOLD OF (10),
cont>     PARTITION AREA1),
cont>     AREA2 (THRESHOLDS ARE (30, 50, 98),
cont>     PARTITION AREA2),
cont>     AREA3 (PARTITION AREA3)
cont>   );
SQL> --
SQL> CREATE STORAGE MAP M3 FOR FIELD_NOTES
cont>   THRESHOLDS OF (50,70,90)
cont>   STORE COLUMNS (WEATHER, TIDE, SIGHTING_TIME)
cont>   VERTICAL PARTITION WEATHER_TIDE_SIGHTINGTIME
cont>   USING (SPECIES_NUMBER, NOTES_NUMBER)
cont>     IN AREA1
cont>       (THRESHOLDS OF (20, 80, 90) )
cont>       WITH LIMIT OF (30, 88)
cont>     IN AREA2
cont>       WITH LIMIT OF (40, 89)
cont>     IN AREA3
cont>       WITH LIMIT OF (50, 90)
cont>     OTHERWISE IN AREA4
cont>       (THRESHOLDS ARE (20, 30, 40))
cont>   STORE COLUMNS (NOTES, NOTES_NUMBER, SPECIES_NUMBER)
cont>   VERTICAL PARTITION NOTES_NOTESNUM_SPECIESNUM
cont>   USING (SPECIES_NUMBER)
cont>     IN AREA5
cont>       (THRESHOLDS OF (20, 80, 90) )
cont>       WITH LIMIT OF (30)
cont>     IN AREA6
cont>       WITH LIMIT OF (40)
cont>     IN AREA7
cont>       WITH LIMIT OF (50)
cont>     OTHERWISE IN AREA8
cont>       (THRESHOLDS ARE (20, 30, 40));
```

CREATE STORAGE MAP Statement

Example 6: Creating a storage map for a table containing data

```
SQL> -- Create table, insert data, and then create a storage map.
SQL> --
SQL> CREATE TABLE MAP_TEST2 (a INTEGER, b CHAR(10));
SQL> INSERT INTO MAP_TEST2 (a, b) VALUES (2, 'Second');
1 row inserted
SQL> CREATE STORAGE MAP MAP_TEST2_MAP FOR MAP_TEST2
cont>     STORE IN RDB$SYSTEM;
SQL> INSERT INTO MAP_TEST2 (a, b) VALUES (22, 'Second2');
1 row inserted
SQL> COMMIT;
SQL> SELECT *,DBKEY FROM MAP_TEST2;
          A  B                               DBKEY
          2  Second                          90:809:0
          22 Second2                         90:809:1
2 rows selected
SQL>
SQL> -- Now alter the storage map and
SQL> -- place it in a different storage area.
SQL>
SQL> ALTER STORAGE MAP MAP_TEST2_MAP
cont>     STORE IN TEST_AREA2;
SQL> COMMIT;
SQL> SELECT *,DBKEY FROM MAP_TEST2;
          A  B                               DBKEY
          2  Second                          91:11:0
          22 Second2                         91:11:1
2 rows selected
SQL>
```

Example 7: Invalid attempts to create a storage map

```
SQL> -- Create table, insert data, and then
SQL> -- create a storage map with invalid attributes.
SQL>
SQL> CREATE TABLE MAP_TEST3 (a INTEGER, b CHAR(10));
SQL> CREATE INDEX MAP_TEST3_INDEX ON MAP_TEST3 (a);
SQL> INSERT INTO MAP_TEST3 (a, b) VALUES (3, 'Third');
1 row inserted
```

CREATE STORAGE MAP Statement

```
SQL>
SQL> CREATE STORAGE MAP MAP_TEST3_MAP FOR MAP_TEST3
cont>     STORE IN TEST_AREA1;           -- Must be the default area.
%RDB-E-NO_META_UPDATE, metadata update failed
-RDMS-F-RELNOTEMPTY, table "MAP_TEST3" has data in it
-RDMS-E-NOCMPLXMAP, can not use complex map for non-empty table
SQL>
SQL> CREATE STORAGE MAP MAP_TEST3_MAP for MAP_TEST3
cont>     PLACEMENT VIA INDEX MAP_TEST3_INDEX  -- Can't use placement.
cont>     STORE IN RDB$SYSTEM;
%RDB-E-NO_META_UPDATE, metadata update failed
-RDMS-F-RELNOTEMPTY, table "MAP_TEST3" has data in it
-RDMS-E-NOCMPLXMAP, can not use complex map for non-empty table
SQL>
SQL> CREATE STORAGE MAP MAP_TEST3_MAP FOR MAP_TEST3
cont>     DISABLE COMPRESSION             -- Can't change compression.
cont>     STORE IN RDB$SYSTEM;
%RDB-E-NO_META_UPDATE, metadata update failed
-RDMS-F-RELNOTEMPTY, table "MAP_TEST3" has data in it
-RDMS-E-NOCMPLXMAP, can not use complex map for non-empty table
SQL>
SQL> CREATE STORAGE MAP MAP_TEST3_MAP for MAP_TEST3
cont>     THRESHOLDS ARE (50, 60, 70)      -- Can't change thresholds.
cont>     STORE IN RDB$SYSTEM;
%RDB-E-NO_META_UPDATE, metadata update failed
-RDMS-F-RELNOTEMPTY, table "MAP_TEST3" has data in it
-RDMS-E-NOCMPLXMAP, can not use complex map for non-empty table
SQL>
SQL> CREATE STORAGE MAP MAP_TEST3_MAP FOR MAP_TEST3
cont>     STORE ACROSS (RDB$SYSTEM, TEST_AREA2);-- Can't use more than one area.
%RDB-E-NO_META_UPDATE, metadata update failed
-RDMS-F-RELNOTEMPTY, table "MAP_TEST3" has data in it
-RDMS-E-NOCMPLXMAP, can not use complex map for non-empty table
SQL>
SQL> CREATE STORAGE MAP MAP_TEST3_MAP for MAP_TEST3
cont>     STORE COLUMNS (a) in RDB$SYSTEM   -- Can't vertically partition.
cont>     STORE COLUMNS (b) in TEST_AREA2;
%RDB-E-NO_META_UPDATE, metadata update failed
-RDMS-F-RELNOTEMPTY, table "MAP_TEST3" has data in it
-RDMS-E-NOCMPLXMAP, can not use complex map for non-empty table
```

CREATE STORAGE MAP Statement

Example 8: Using the RMU Extract command to display WITH LIMIT OF expressions

The WITH LIMIT OF clauses of the STORE clause are converted to Boolean expressions that are used by Oracle Rdb to direct inserted data to the correct storage area. You can use the RMU Extract command to display these Boolean expressions. Use the Item=STORAGE_MAP and Option=FULL qualifiers as shown in the following example.

```
$ RMU/EXTRACT-
_ $ /ITEM=STORAGE_MAP-
_ $ /OPTION=(MATCH:EMPLOYEES_MAP%,NOHEADER,FULL,FILENAME_ONLY) -
_ $ DB$:MF_PERSONNEL
set verify;
set language ENGLISH;
set default date format 'SQL92';
set quoting rules 'SQL92';
set date format DATE 001, TIME 001;
attach 'filename MF_PERSONNEL.RDB';
create storage map EMPLOYEES_MAP
  for EMPLOYEES
  comment is
    ' employees partitioned by "00200" "00400"'
  placement via index EMPLOYEES_HASH
  store
    using (EMPLOYEE_ID)
    -- Partition:
    --      (EMPLOYEE_ID <= '00200')
    in EMPIDS_LOW
    with limit of ('00200')
    -- Partition:
    --      (EMPLOYEE_ID <= '00400')
    in EMPIDS_MID
    with limit of ('00400')
    otherwise in EMPIDS_OVER;
commit work;
```

Example 9: SQL Mapping Routine

This example shows the SQL mapping routine created by the CREATE STORAGE MAP statement that matches the WITH LIMIT OF clause for the storage map.

CREATE STORAGE MAP Statement

```
SQL> create table EMPLOYEES (  
cont>     EMPLOYEE_ID      CHAR (5),  
cont>     LAST_NAME         CHAR (14),  
cont>     FIRST_NAME         CHAR (10),  
cont>     MIDDLE_INITIAL      CHAR (1),  
cont>     ADDRESS_DATA_1     CHAR (25),  
cont>     ADDRESS_DATA_2     CHAR (25),  
cont>     CITY                 CHAR (20),  
cont>     STATE                CHAR (2),  
cont>     POSTAL_CODE         CHAR (5),  
cont>     SEX                  CHAR (1),  
cont>     BIRTHDAY            DATE VMS,  
cont>     STATUS_CODE         CHAR (1));  
SQL>  
SQL>     create storage map EMPLOYEES_MAP  
cont>         for EMPLOYEES  
cont>         comment is  
cont>         ' employees partitioned by "00200" "00400" '  
cont>         store  
cont>         using (EMPLOYEE_ID)  
cont>             in EMPIDS_LOW  
cont>                 with limit of ('00200')  
cont>             in EMPIDS_MID  
cont>                 with limit of ('00400')  
cont>             otherwise in EMPIDS_OVER;  
SQL>  
SQL> commit work;  
SQL>  
SQL> show system modules;  
Modules in database with filename MF_PERSONNEL  
RDB$STORAGE_MAPS  
SQL>  
SQL> show system functions;  
Functions in database with filename MF_PERSONNEL  
EMPLOYEES_MAP  
SQL>  
SQL> show system function EMPLOYEES_MAP;  
Information for function EMPLOYEES_MAP  
  
Function ID is: -2  
Source:  
return  
    case  
        when (:EMPLOYEE_ID <= '00200') then 1  
        when (:EMPLOYEE_ID <= '00400') then 2  
        else 3  
    end case;  
Comment:      Return value for select partition - range 1 .. 3  
Module name is: RDB$STORAGE_MAPS  
Module ID is: -1  
Number of parameters is: 1
```

CREATE STORAGE MAP Statement

Parameter Name	Data Type	Domain or Type
-----	-----	-----
	INTEGER	
Function result datatype		
Return value is passed by value		
EMPLOYEE_ID	CHAR(5)	
Parameter position is 1		
Parameter is IN (read)		
Parameter is passed by reference		

CREATE SYNONYM Statement

CREATE SYNONYM Statement

Creates an alternate name or synonym for an existing database object. The object may be a domain, function, module, procedure, sequence, another synonym, table, or view.

Once defined, the synonym can be used in any query or data definition language statement in place of the referenced object.

However, the SHOW commands do not accept synonyms. Use the SHOW SYNONYM statement to determine if the name is a synonym.

Environment

You can use the CREATE SYNONYM statement:

- In interactive SQL
- Embedded in host language programs
- As part of a procedure in an SQL module or other compound statement
- In dynamic SQL as a statement to be dynamically executed

Format

```
CREATE OR REPLACE PUBLIC
  SYNONYM <synonym-name> FOR object-type
  <object-name> COMMENT IS '<quoted-string>'
```

object-type =

```
→ DOMAIN →
→ FUNCTION →
→ MODULE →
→ PROCEDURE →
→ SEQUENCE →
→ SYNONYM →
→ TABLE →
→ VIEW →
```


CREATE SYNONYM Statement

Arguments

COMMENT IS 'quoted-string'

This optional clause can be used to add several lines of comment to the synonym object. The comment is displayed by the SHOW SYNONYM statement.

FOR object-name

The name of the database object for which the synonym is required. This name must exist for an object in the database. If the optional object type is omitted, then Oracle Rdb will search the database for an object with this name.

DOMAIN
FUNCTION
MODULE
PROCEDURE
SEQUENCE
SYNONYM
TABLE

These optional object types can be used when the referenced object name is not unique within the database. For instance, Oracle Rdb allows a domain and a table to both be called MONEY. Therefore, to create a synonym for the table MONEY, you must use the FOR TABLE clause so that it is uniquely identified.

OR REPLACE

Instructs SQL to replace any synonym of this name if it exists. If it does not exist, a new synonym is created. This shorthand allows replacement of an existing synonym while maintaining all the dependencies established by query and DDL usage of this synonym.

PUBLIC

This optional clause is provided for compatibility with the Oracle database server. It is currently not used by Oracle Rdb. Its presence or absence may be used by future releases. Oracle Corporation recommends you use the PUBLIC keyword in applications.

synonym-name

The name of the synonym you want to create. The synonym name must be unique within all domains, tables, views, functions, procedures, modules, sequences, and synonyms within the database. You may qualify it with an alias.

CREATE SYNONYM Statement

Usage Notes

- You must have the database `CREATE` privilege to execute the `CREATE SYNONYM` statement.
- You must have the `REFERENCES` privilege on the referenced object to create a synonym for that object. Because domains do not have access control, no other privileges are required to create synonyms for domains.
- The database must have synonyms enabled. The `ALTER DATABASE . . . SYNONYMS ARE ENABLED` clause creates a new system relation, `RDB$OBJECT_SYNONYMS`, which is used to record the synonyms created by this statement.

- Synonyms do not have any access control. Instead, granting privileges to, or revoking privileges from a synonym is the same as referencing the base object. In the following example, the `GRANT` statement grants the `SELECT` privilege to `PUBLIC` on the `EMPLOYEES` table:

```
SQL> CREATE SYNONYM EMPS FOR EMPLOYEES;  
SQL> GRANT SELECT ON TABLE EMPS TO PUBLIC;
```

- You may create synonyms for synonyms. This forms a chain of synonyms that must be processed to determine the base database object. Oracle Corporation recommends that this chain be no more than 10 references. Oracle Rdb enforces a chain maximum length of 64.

Examples

Example 1: Using the Default Alias

```
SQL> CREATE SYNONYM emps FOR employees;
```

Example 2: Using an Explicit Alias for the Synonym

```
SQL> CREATE SYNONYM db1.emps FOR employees;
```

Example 3: Using an Explicit Alias for the Referenced Object

```
SQL> CREATE SYNONYM emps FOR db1.employees;
```

Example 4: Using the Alias Explicitly

```
SQL> CREATE SYNONYM db1.emps FOR db1.employees;
```

CREATE SYNONYM Statement

Example 5: Using the Table Type

```
SQL> CREATE SYNONYM cash FOR table money
cont> COMMENT IS 'use a different name to avoid confusion with'
cont> / 'the domain MONEY';
```

Example 6: Using Multiple Synonyms

```
SQL> CREATE TABLE t_employees_0001 (...);
SQL> CREATE SYNONYM employees FOR t_employees_0001;
SQL> CREATE SYNONYM emps FOR employees;
```

CREATE TABLE Statement

CREATE TABLE Statement

Creates a temporary or persistent base table definition. A table definition consists of a list of definitions of columns that make up a row in the table.

Persistent base tables are tables whose metadata and data are stored in the database beyond an SQL session. The data can be shared by all users attached to the database.

Temporary tables are tables whose data is automatically deleted when an SQL session or module ends. The tables only materialize when you refer to them in an SQL session and the data is local to an SQL session. You can also specify whether the data is preserved or deleted at the end of a transaction within the session; the default is to delete the data. The data in temporary tables is private to the user. There are three types of temporary tables:

- Global temporary tables
- Local temporary tables
- Declared local temporary tables (see the `DECLARE LOCAL TEMPORARY TABLE` Statement for additional information)

The metadata for a global temporary table is stored in the database and persists beyond the SQL session. Different SQL sessions can share the same metadata. The data stored in the table cannot be shared between SQL sessions. However, the data can be shared between modules in a single SQL session. The data does not persist beyond an SQL session.

The metadata for a local temporary table is stored in the database and persists beyond the SQL session. Different SQL sessions can share the same metadata. The data stored in the table cannot be shared between different modules in a single SQL session or between SQL sessions. The data does not persist beyond an SQL session or module.

Because temporary tables are used only to hold the user's data, which is not shared among users, no locks are needed and the data can be modified in a read-only transaction.

See the *Oracle Rdb Guide to Database Design and Definition* for more information on temporary tables.

Information tables are special read-only tables that can be used to retrieve database attributes that are not stored in the existing relational tables. Information tables allow interesting database information, which is currently stored in an internal format, to be displayed as a relational table.

CREATE TABLE Statement

When you define a table, you can also define table constraints. A **constraint** specifies a condition that restricts the values that can be stored in a table. Constraints can specify that columns contain:

- Only certain values
- Primary key values
- Unique values
- Values that cannot be null

There are several ways to specify a table definition in the CREATE TABLE statement:

- Directly by naming the table, its columns and associated data types, default values (optional), constraint definitions (optional), and formatting clauses
You can define constraints on persistent base tables and global temporary tables only.
- Indirectly by providing a path name for a repository record definition that specifies the table name, columns, and data types
- Indirectly by providing another table as a model in inheriting the columns, datatypes and NOT NULL constraints.

SQL allows you to specify the default character data type or the national character data type when defining table columns.

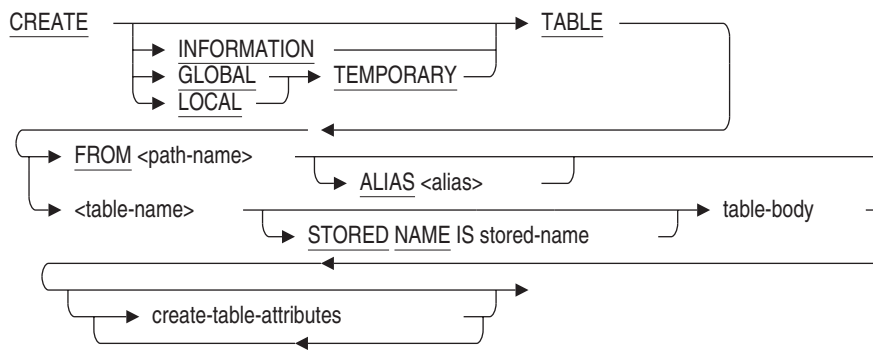
Environment

You can use the CREATE TABLE statement:

- In interactive SQL
- Embedded in host language programs to be precompiled
- As part of a procedure in an SQL module
- In dynamic SQL as a statement to be dynamically executed

CREATE TABLE Statement

Format



create-table-attributes =

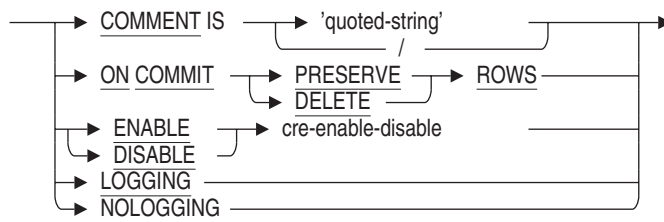
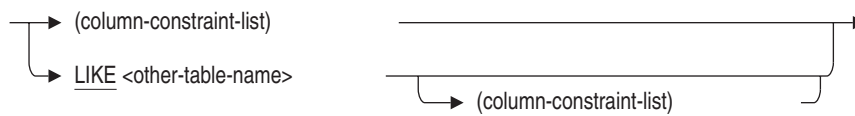
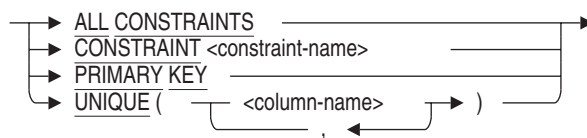


table-body =

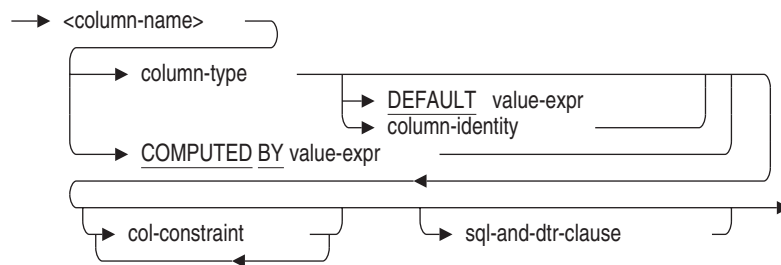


cre-enable-disable =



CREATE TABLE Statement

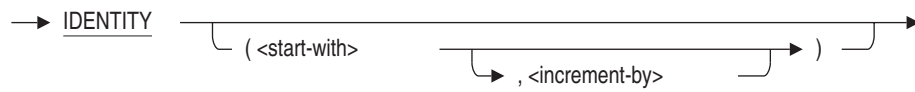
col-definition =



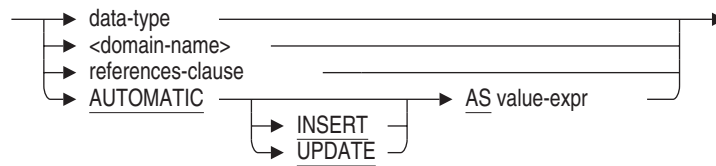
column-constraint-list =



column-identity =

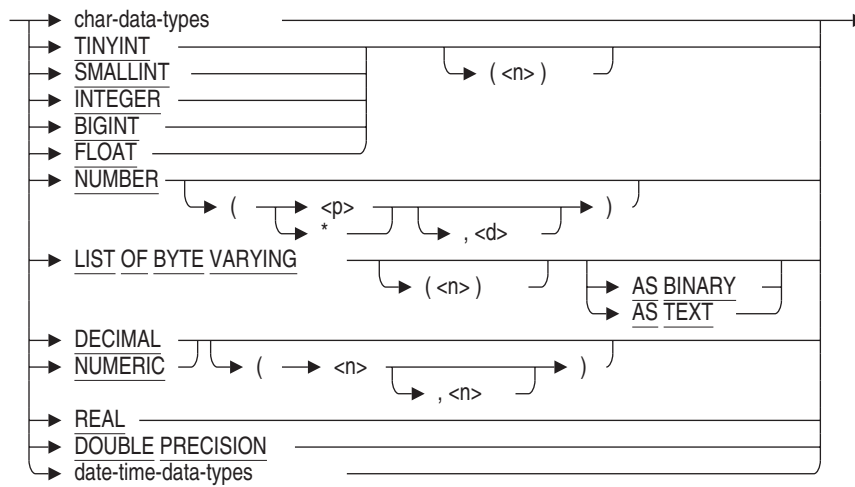


column-type =

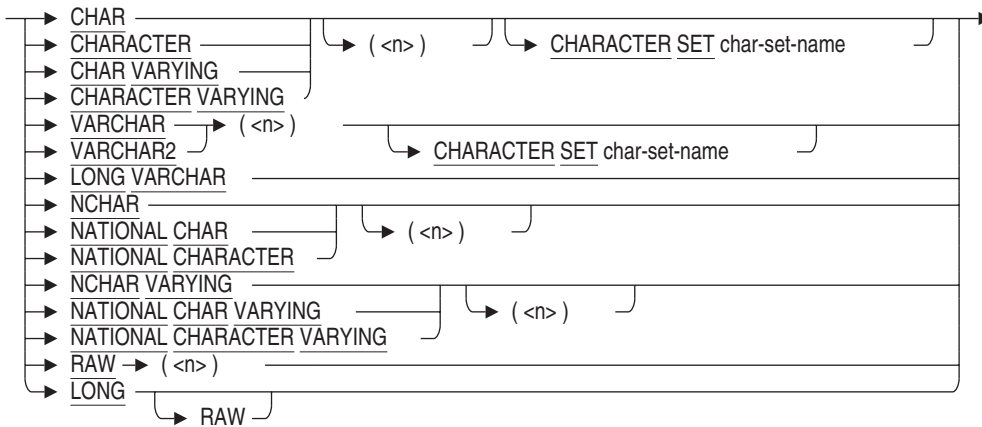


CREATE TABLE Statement

data-type =

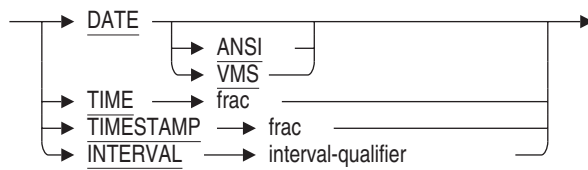


char-data-types =

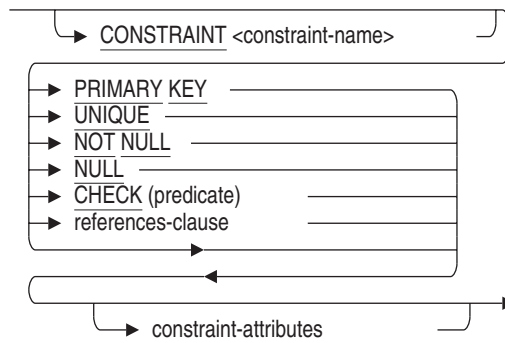


CREATE TABLE Statement

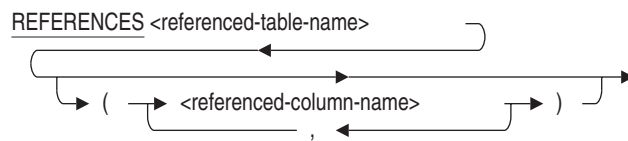
date-time-data-types =



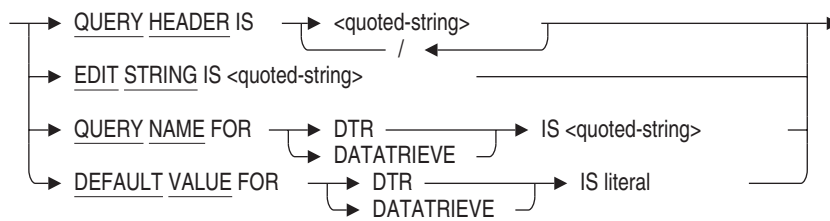
col-constraint=



references-clause =



sql-and-dtr-clause =



CREATE TABLE Statement

literal =

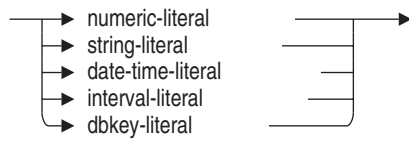


table-constraint =

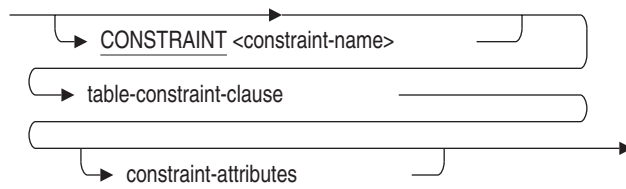
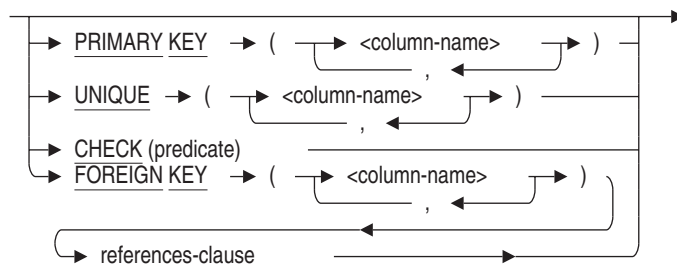
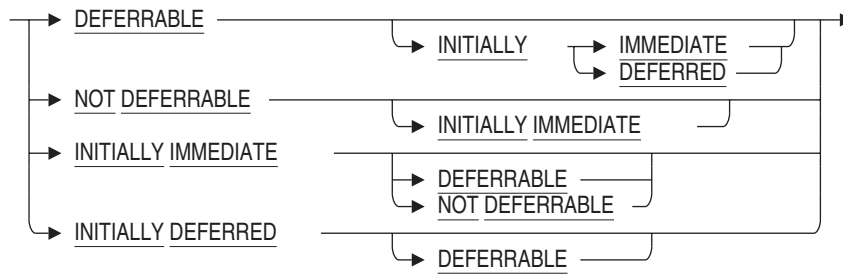


table-constraint-clause =



constraint-attributes =



CREATE TABLE Statement

Arguments

ALIAS alias

Specifies a name for an attach to a particular database. SQL adds the table definition to the database referred to by the alias.

If you do not specify an alias, SQL adds the table definition to the default database. See Section 2.2.1 for more information on default databases and aliases.

AUTOMATIC AS value-expr

AUTOMATIC INSERT AS value-expr

AUTOMATIC UPDATE AS value-expr

These AUTOMATIC column clauses allow you to store special information when data is inserted into a row or a row is updated. For example, you can log application-specific information to audit activity or provide essential values, such as time stamps or unique identifiers for the data.

The assignment of values to these types of columns is managed by Oracle Rdb. The AUTOMATIC INSERT clause can be used to provide a complex default for the column when the row is inserted; it cannot be changed by an UPDATE statement. The AUTOMATIC UPDATE clause can be used to provide an updated value during an UPDATE statement. The unqualified AUTOMATIC clause specifies that the value expression should be applied during both INSERT and UPDATE statements. The column type is derived from the AS value-expr; using CAST allows a specific data type to be specified. However, this is not required and is rarely necessary.

You can define an AUTOMATIC INSERT column to automatically receive data during an insert operation. The data is stored like any other column, but the column is read-only. Because AUTOMATIC columns are treated as read-only columns, they cannot appear in the column list for an insert operation nor be modified by an update operation. AUTOMATIC UPDATE columns can have an associated default value that will be used when the row is inserted. See Example 15 in the Examples section.

char-data-type

A valid SQL character data type. See Section 2.3.1 for more information on character data types.

character-set-name

A valid character set name.

CREATE TABLE Statement

CHECK predicate

Specifies a predicate that column values inserted into the table must satisfy. See Section 2.7 for details on specifying predicates.

Predicates in CHECK column constraints can refer directly only to the column with which they are associated. See the Usage Notes for details.

col-constraint

A constraint that applies to values stored in the associated column.

SQL allows column constraints and table constraints. The Usage Notes summarize the differences between column constraints and table constraints. The five types of column constraints are PRIMARY KEY, UNIQUE, NOT NULL, CHECK, and FOREIGN KEY constraints. The FOREIGN KEY constraints are created with the REFERENCES clause.

You can define a column constraint on persistent base tables and global temporary tables only.

col-definition

The definition for a column in the table. SQL gives you two ways to specify column definitions:

- By directly specifying a data type to associate with a column name
- By naming a domain that indirectly specifies a data type to associate with a column name

Either way also allows options for specifying default values, column constraints, and formatting clauses.

column-name

The name of a column you want to create in the table. You need to specify a column name whether you directly specify a data type in the column definition or indirectly specify a data type by naming a domain in the column definition.

COMPUTED BY value-expr

Specifies that the value of this column is calculated from values in other columns and constant expressions.

If your column definition refers to a column name within a value expression, that named column must already be defined within the same CREATE TABLE statement. See Section 2.6 for information on value expressions.

Any column that you refer to in the definition of a computed column cannot be deleted from that table unless you first delete the computed column.

CREATE TABLE Statement

SQL does not allow the following for computed columns:

- UNIQUE constraints
- REFERENCES clauses
- PRIMARY KEY constraints
- DEFAULT clause
- IDENTITY clause
- Default value for DATATRIEVE

For example, if the FICA_RATE for an employee is 6.10 percent of the employee's starting salary and the group insurance rate is 0.7 percent, you can define FICA_RATE and GROUP_RATE columns like this:

```
SQL> CREATE TABLE payroll_detail
cont> (salary_code CHAR(1),
cont> starting_salary SMALLINT(2),
cont> fica_amt
cont>     COMPUTED BY (starting_salary * 0.061),
cont> group_rate
cont>     COMPUTED BY (starting_salary * 0.007));
```

When you use this type of definition, you only have to store values in the salary_code and starting_salary columns. The FICA and group insurance deduction columns are computed automatically when the columns fica_amt or group_rate are selected.

Example 11 shows a COMPUTED BY column that uses a select expression.

constraint-attributes

Although the constraint attribute syntax, shown in Table 6-3, provides 11 permutations as required by the SQL99 standard, they equate to the following three options:

- INITIALLY IMMEDIATE NOT DEFERRABLE
Specifies that evaluation of the constraint must take place when the INSERT, DELETE, or UPDATE statement executes. If you are using the SQL99, SQL92, MIA, ORACLE LEVEL1, or ORACLE LEVEL2 dialect, this is the default.
- INITIALLY IMMEDIATE DEFERRABLE
Specifies that evaluation of the constraint may be deferred (using the SET CONSTRAINT ALL statement or the SET TRANSACTION statement with the EVALUATING clause), but by default it is evaluated after the

CREATE TABLE Statement

INSERT, DELETE, or UPDATE statement executes. See the SET ALL CONSTRAINTS Statement for more information.

- **INITIALLY DEFERRED DEFERRABLE**

Specifies that evaluation of the constraint can take place at any later time. Unless otherwise specified, evaluation of the constraint takes place as the COMMIT statement executes. You can use the SET ALL CONSTRAINTS statement to have all constraints evaluated earlier. See the description of the SET ALL CONSTRAINTS statement for more information.

If you are using the default SQLV40 dialect, this is the default constraint attribute. When using this dialect, Oracle Rdb displays a deprecated feature message for all constraints defined without specification of one of the constraint attributes.

CONSTRAINT constraint-name

Specifies a name for a column or table constraint. The name is used for a variety of purposes:

- The RDB\$INTEG_FAIL error message specifies the name when an INSERT, UPDATE, or DELETE statement violates the constraint.
- The ALTER TABLE table-name DROP CONSTRAINT constraint-name statement specifies the name to delete a table constraint.
- The SHOW TABLE statements display the names of column and table constraints.
- The EVALUATING clause of the SET TRANSACTION and DECLARE TRANSACTION statements specifies constraint names.
- The ENABLE and DISABLE clauses of the ALTER and CREATE TABLE statements specify constraint names.
- The ALTER CONSTRAINT statement specifies constraint names.
- The DROP CONSTRAINT statement

The CONSTRAINT clause is optional. If you omit the constraint name, SQL creates a name. However, Oracle Rdb recommends that you always name column and table constraints. If you supply a constraint name with the CONSTRAINT clause, it must be unique in the database or in the schema if you are using a multischema database.

data-type

A valid SQL data type. Specifying an explicit data type to associate with a column is an alternative to specifying a domain name. See Section 2.3 for more information on data types.

CREATE TABLE Statement

date-time-data-types

A data type that specifies a date, time, or interval. See Section 2.3.2 for more information about date-time data types.

DEFAULT value-expr

Provides a default value for a column if the row that is inserted does not include a value for that column.

You can use any value expression including subqueries, conditional, character, date/time, and numeric expressions as default values. See Section 2.6 for more information about value expressions.

For more information about NULL, see Section 2.6.1 and the Usage Notes following this Arguments list.

The value expressions described in Section 2.6 include DBKEY and aggregate functions. However, the DEFAULT clause is not a valid location for referencing a DBKEY or an aggregate function. If you attempt to reference either, you receive a compile-time error.

If you do not specify a default value, a column inherits the default value from the domain. If you do not specify a default value for either the column or domain, SQL assigns NULL as the default value.

domain-name

The name of a domain created in a CREATE DOMAIN statement. SQL gives the column the data type specified in the domain. For more information on domains, see the CREATE DOMAIN Statement.

For most purposes, you should specify a domain instead of an explicit data type.

- Domains ensure that all columns in multiple tables that serve the same purpose have the same data type. For example, several tables in the sample personnel database refer to the domain ID_DOM.
- A domain lets you change the data type for all columns that refer to it in one operation by changing the domain itself with an ALTER DOMAIN statement.

For example, if you want to change the data type for the column EMPLOYEE_ID from CHAR(5) to CHAR(6), you need only alter the data type for the domain ID_DOM. You do not have to alter the data type for the column EMPLOYEE_ID in the tables DEGREES, EMPLOYEES, JOB_HISTORY, or SALARY_HISTORY, nor do you have to alter the column MANAGER_ID in the DEPARTMENTS table.

CREATE TABLE Statement

However, you might not want to use domains when you create tables if:

- Your application must be compatible with Oracle RDBMS.
- You are creating intermediate result tables that do not need the advantages of domains.

enable-disable-clause

Allows you to enable or disable all constraints, specified constraints, a primary key, or a unique column name, as described in the following list. By default, table and column constraints added during a create table operation are enabled.

- **DISABLE ALL CONSTRAINTS**
All table and column constraints for this table are disabled. No error is raised if no constraints are defined on the table.
- **ENABLE ALL CONSTRAINTS**
All and column constraints for this table are enabled. No error is raised if no constraints are defined on the table.
- **DISABLE CONSTRAINT constraint-name**
The named constraint is disabled. The named constraint must be a table or column constraint for the table.
- **ENABLE CONSTRAINT constraint-name**
The named constraint is enabled. The named constraint must be a table or column constraint for the table.
- **DISABLE PRIMARY KEY**
The primary key for the table is disabled.
- **ENABLE PRIMARY KEY**
The primary key for the table is enabled.
- **DISABLE UNIQUE (column-name)**
The matching **UNIQUE** constraint is disabled. The columns listed must be columns in the table.
- **ENABLE UNIQUE (column-name)**
The matching **UNIQUE** constraint is enabled. The columns listed must be columns in the table.

CREATE TABLE Statement

FOREIGN KEY column-name

The name of a column or columns that you want to declare as a foreign key in the table you are defining (referencing table). You cannot declare a computed column as a foreign key.

FROM path-name

Specifies the repository path name of a repository record definition. SQL creates the table using the definition from this record and gives the table the name of the record definition.

You can create a table using the FROM path-name clause only if the record definition in the repository was originally created using the repository Common Dictionary Operator (CDO) utility. For instance, you cannot create a table using the FROM path-name clause if the record definition was created in the repository as part of an SQL session.

If the repository record contains a nested record definition, you cannot create a table based on it.

Creating a table based on a repository record definition is useful when many applications share the same definition. Changes to the common definition can be automatically reflected in all applications that use it.

Note

Changes by other users or applications to the record definition in the repository affect the table definition once the database is integrated to match the repository with an `INTEGRATE DATABASE . . . ALTER FILES` statement. If those changes include deleting records or fields on which tables or table columns are based, any data in the dependent table or table column is lost after the next `INTEGRATE DATABASE . . . ALTER FILES` statement executes.

You can use the FROM clause only if the database was attached specifying `PATHNAME`. You can specify either a full repository path name or a relative repository path name.

You cannot define constraints or any other table definition clauses, such as `DATATRIEVE` formatting clauses, when you use the FROM path-name form of the `CREATE TABLE` statement. This restriction does not prevent you from using an `ALTER TABLE` statement to add them later.

You cannot use the FROM path-name clause when embedding a `CREATE TABLE` statement within a `CREATE DATABASE` statement.

CREATE TABLE Statement

GLOBAL TEMPORARY

LOCAL

Specifies that the table definition is either a global or local temporary table.

IDENTITY

Specifies that the column is to be a special read-only identity column. `INSERT` will evaluate this column and store a unique value for each row inserted.

Only one column of a table may have the `IDENTITY` attribute. Rdb creates a sequence with the same name as the current table.

See `ALTER SEQUENCE` Statement and `CREATE SEQUENCE` Statement for more information.

increment-by

An integer literal value that specifies the increment for the sequence created for the `IDENTITY` column. A negative value creates a descending sequence, and a positive value creates an ascending sequence. A value of zero is not permitted. If omitted the default is 1, that is an ascending sequence.

INFORMATION

Specifies that the table definition is an information table. For details on information tables, see Appendix I in Volume 5.

Information tables are reserved for use by Oracle Corporation.

LIKE other-table-name

Allows a database administrator to copy the metadata for an existing table and create a new table with similar characteristics. An optional column list can be used to add an extra columns and constraints to this table. The referenced table must exist in the same database as the table being created.

LOGGING

NOLOGGING

The `LOGGING` clause specifies that the `CREATE TABLE` statement should be logged in the recovery-unit journal file (.ruj) and after-image journal file (.aij).

The `NOLOGGING` clause specifies that the `CREATE TABLE` statement should not be logged in the recovery-unit journal file (.ruj) and after-image journal file (.aij).

The `LOGGING` clause is the default.

NOT NULL

Restricts values in the column to values that are not null.

CREATE TABLE Statement

ON COMMIT PRESERVE ROWS

ON COMMIT DELETE ROWS

Specifies whether data is preserved or deleted after a COMMIT statement for global or local temporary tables only.

The default, if not specified, is ON COMMIT DELETE ROWS.

PRIMARY KEY

A primary key constraint defines one or more columns whose values make a row in a table different from all others. SQL requires that values in a primary key column be unique and not null; therefore, you need not specify the UNIQUE and NOT NULL column constraints for primary key columns.

You cannot specify the primary key constraint for a computed column.

When used as a table constraint this clause must be followed by a list of column names. When used as a column constraint this clause applies to the named column of the table.

references-clause

Specifies the name of the column or columns that are a unique key or primary key or in the referenced table. When the REFERENCES clause is used as a table constraint, the column names specified in the FOREIGN KEY clause become a foreign key for the referencing table.

When used as the column type clause, specifies that the type of the column be inherited from the PRIMARY KEY or UNIQUE index referenced. Both the data type and domain are inherited.

REFERENCES referenced-table-name

Specifies the name of the table that contains the unique key or primary key referenced by the referencing table. To declare a constraint that refers to a unique or primary key in another table, you must have the SQL REFERENCES or CREATE privileges to the referenced table.

referenced-column-name

For a column constraint, the name of the column that is a unique key or primary key in the referenced table. You cannot use a computed column as a referenced column name. For a table constraint, the referenced column name is the name of the column or columns that are a unique key or primary key in the referenced table. If you omit the referenced-column-name clause, the primary key is selected by default. The number of columns and their data types must match.

CREATE TABLE Statement

sql-and-dtr-clause

Optional SQL formatting clause. See Section 2.5 for more information about formatting clauses.

If you specify a formatting clause for a column that is based on a domain that also specifies a formatting clause, the formatting clause in the table definition overrides the one in the domain definition.

start-with

An integer literal value that specifies the starting value for the sequence created for the IDENTITY column. If omitted the default is 1.

STORED NAME IS stored-name

Specifies a name that Oracle Rdb uses to access a table created in a multischema database. The stored name allows you to access multischema definitions using interfaces, such as Oracle RMU, the Oracle Rdb management utility, that do not recognize multiple schemas in one database. You cannot specify a stored name for a table in a database that does not allow multiple schemas. For more details about stored names, see Section 2.2.18.

table-constraint

A constraint definition that applies to the whole table.

SQL allows column constraints and table constraints. The Usage Notes summarize the differences between the two types of constraints. The four types of table constraints are PRIMARY KEY, UNIQUE, CHECK, and FOREIGN KEY constraints.

A column must be defined in a table before you can specify the column in a table constraint definition.

You can define a table constraint on persistent base tables and global temporary tables only.

table-name

The name of the table definition you want to create. Use a name that is unique among all table, sequence, view and synonym names in the database, or in the schema if you are using a multischema database. Use any valid SQL name. (See Section 2.2 for more information on user-supplied names.)

UNIQUE

Specifies that values in the associated column must be unique. You can use either the UNIQUE or PRIMARY KEY keywords to define one or more columns as a unique key for a table.

CREATE TABLE Statement

You cannot specify the UNIQUE constraint for a computed column or for a column defined with the LIST OF BYTE VARYING data type.

Usage Notes

- You must have the CREATE database privilege on the database to create a table. You must have REFERENCE database privilege on the table specified by the LIKE clause.
- When the CREATE TABLE statement executes, SQL adds the table definition to the database.
If you declared the database with the PATHNAME specification, the definition is also added to the repository.
- It is possible when using the repository to define record structures that are not acceptable to Oracle Rdb.
The repository is intended as a generic data repository that can hold data structures available to many layered products and languages.
These data structures may not always be valid when applied to the relational data model used by Oracle Rdb.
The following are some of the common incompatibilities between the data structures of the repository and Oracle Rdb.
 - %CDD-E-PRSMISSNG, attribute value is missing
This error can occur when a record definition in the repository contains a VARIANTS clause.
 - %CDD-E-INVALID_RDB_DTY, data type of field is not supported by Oracle Rdb
This error can occur when a record definition in the repository contains an OCCURS clause.
 - %CDD-E-DTYPE_REQUIRED, field must have a data type for inclusion in an Oracle Rdb database
This error can occur when a record definition in the repository contains another nested record definition. Oracle Rdb can only accept field definitions in a record definition.
 - %CDD-E-INVALID_RDB_DIM, record PARTS has dimension and cannot be used by Oracle Rdb
This error occurs when a record definition in the repository contains an ARRAY clause.

CREATE TABLE Statement

- The CREATE TABLE statement creates a default access privilege set for the table that gives the creator all privileges to the database and all other users no privileges. This means that new tables have a PUBLIC access of NONE.

To override default PUBLIC access for newly created tables, define an identifier with the name DEFAULT in the system privileges table. The access rights that you give to this identifier on your database will then be assigned to any new tables that you create.

1. Assigning the SELECT and UPDATE privileges to the database with alias TEST1

```
SQL> ATTACH 'ALIAS test1 FILENAME mf_personnel';
SQL> SHOW PROTECTION ON DATABASE test1;
Protection on Alias TEST1
  (IDENTIFIER=[DBS, SMALLWOOD], ACCESS=SELECT+INSERT+UPDATE+DELETE+
    SHOW+CREATE+ALTER+DROP+DBCTRL+OPERATOR+DBADM+SECURITY+DISTRIBTRAN)
  (IDENTIFIER=[*, *], ACCESS=NONE)
SQL> GRANT SELECT, UPDATE ON DATABASE ALIAS TEST1
cont> TO DEFAULT;
```

2. Committing and disconnecting the transaction to make the change in protection occur

```
SQL> COMMIT;
SQL> DISCONNECT ALL;
```

3. Receiving all access rights to the new table TABLE1

The protection on existing tables in the database is not changed; however, any new tables that you define receive the protection specified by the DEFAULT identifier. In this example, the owner (SMALLWOOD) receives all the access rights to the new table TABLE1, and all other users receive the SELECT and UPDATE access rights specified by the DEFAULT identifier.

```
SQL> ATTACH 'ALIAS test1 FILENAME mf_personnel';
SQL> SET TRANSACTION READ WRITE;
SQL> CREATE TABLE test1.table1
cont>   (last_name_dom CHAR(5),
cont>   year_dom SMALLINT);
SQL> SHOW PROTECTION ON test1.table1;
Protection on Table TEST1.TABLE1
  (IDENTIFIER=[DBS, SMALLWOOD], ACCESS=SELECT+INSERT+UPDATE+DELETE+
    SHOW+CREATE+ALTER+DROP+DBCTRL+REFERENCES)
  (IDENTIFIER=[*, *], ACCESS=SELECT+UPDATE)
```

CREATE TABLE Statement

The DEFAULT identifier is typically present on an OpenVMS system. However, if the DEFAULT identifier has been removed from your system, Oracle Rdb returns an error message.

```
SQL> GRANT INSERT ON DATABASE ALIAS TEST1 to DEFAULT;  
%SYSTEM-F-NOSUCHID, unknown rights identifier
```

- You should consider what value, if any, you want to use for the default value for a column. You can use a value such as NULL or Not Applicable that clearly demonstrates that no data was inserted into a column. If a column usually contains a particular value, you can use that value as the default. For example, if most company employees work full-time, you could make full-time the default value for a work status column.
- If you specify a default value for a column that you base on a domain and you have specified a default value for that domain, the default value for the column overrides the default value for the domain.
- Table-specific constraints can be declared at the table level or the column level or both. These constraints can specify that columns contain only certain values, primary key values, unique values, or that values cannot be missing (null). Multiple constraints can be declared at both the table and column level.

On both levels, you can specify definitions of unique, primary, and foreign keys, and foreign key references to unique or primary keys. You can also specify constraint evaluation time (either commit or update).

On the table level, you can define constraints for multicolumn keys.

On the column level, you can restrict the values of columns to values that are not null.

- You can control when the database system evaluates constraints using the SET ALL CONSTRAINTS statement.
- If you defined constraints as NOT DEFERRABLE, they must be evaluated when the INSERT, DELETE, or UPDATE statement executes. You cannot use either the SET ALL CONSTRAINTS statement or the SET TRANSACTION EVALUATING statement to change the evaluation time.
- Constraints specify a condition that restricts the values that can be stored in a table. The INSERT, UPDATE, or DELETE statements that violate the condition fail. The database system generates an RDB\$_INTEG_FAIL error, and SQL returns an SQLCODE value of -1001.

CREATE TABLE Statement

You can control when the database system evaluates constraints in the EVALUATING clause of DECLARE and SET TRANSACTION statements. By default, all deferred constraints are evaluated when a transaction issues a COMMIT statement. However, if you specify VERB TIME for specific constraints in the EVALUATING clause of a DECLARE or SET TRANSACTION statement, the database system evaluates those constraints whenever UPDATE, INSERT, or DELETE statements execute.

SQL allows column constraints and table constraints. The semantics and syntax for the two types of constraints are similar, but not identical. The following list summarizes the differences:

- Column constraints allow the UNIQUE argument; table constraints allow the UNIQUE (column-name) argument. Specifying UNIQUE for a series of column definitions is more restrictive than specifying UNIQUE and a list of the same columns because SQL requires only that the combination of columns in a UNIQUE (column-name) table constraint be unique.

```
SQL> CREATE TABLE TEMP1
cont> ( COL1 REAL NOT NULL UNIQUE CONSTRAINT C1,
cont>   COL2 REAL NOT NULL UNIQUE CONSTRAINT C2,
cont>   COL3 REAL NOT NULL UNIQUE CONSTRAINT C3 );
SQL>
SQL> CREATE TABLE TEMP2
cont> ( COL4 REAL NOT NULL CONSTRAINT C4,
cont>   COL5 REAL NOT NULL CONSTRAINT C5,
cont>   COL6 REAL NOT NULL CONSTRAINT C6,
cont>   UNIQUE (COL4, COL5, COL6) CONSTRAINT C7 );
SQL>
SQL> INSERT INTO TEMP1 VALUES (1,1,1);
1 row inserted
SQL> INSERT INTO TEMP2 VALUES (1,1,1);
1 row inserted
SQL> COMMIT;
SQL>
SQL> -- This fails because the values
SQL> -- in COL1 will not be unique:
SQL> INSERT INTO TEMP1 VALUES (1,2,2);
1 row inserted
SQL> COMMIT;
%RDB-E-INTEG_FAIL, violation of constraint C1 caused operation to fail
SQL>
```


CREATE TABLE Statement

```
SQL> ROLLBACK;
SQL>
SQL> -- This succeeds because the *combination*
SQL> -- of the columns is still unique:
SQL> INSERT INTO TEMP2 VALUES (1,2,2);
1 row inserted
SQL> COMMIT;
```

- The CHECK constraints have the same syntax for column constraints as for table constraints. The only syntactic distinction between the two CHECK constraints is that CHECK table constraints are separated from column definitions by commas, and CHECK column constraints are not.

The predicate in a CHECK column constraint can refer directly only to the column with which it is associated. The predicate in a CHECK table constraint can refer directly to any column in the table. Either type of CHECK constraint, however, can refer to columns in other tables in the database through column select expressions in the predicate.

The predicate of a CHECK constraint must not be false. It may be unknown. The constraint `COL 10 > 100` would allow values 101, 1000, and NULL. It would not allow the value 99.

```
SQL> -- Cannot directly refer to TEST1 in
SQL> -- column constraint for TEST2:
SQL> CREATE TABLE TEST
cont> ( TEST1 CHAR(5),
cont>   TEST2 CHAR(5)
cont>   CHECK (TEST2 <> TEST1)
cont> );
%SQL-F-COLNOTVAL, The column CHECK constraint cannot refer to the
column TEST1
SQL> -- To get around the problem, make the CHECK constraint a table
SQL> -- constraint by separating it from the column with a comma:
SQL> CREATE TABLE TEST
cont> ( TEST1 CHAR(5),
cont>   TEST2 CHAR(5),
cont>   CHECK (TEST2 <> TEST1)
cont> );
SQL> COMMIT;
SQL> INSERT INTO TEST VALUES ('1','1');
1 row inserted
SQL> COMMIT;
%RDB-E-INTEG_FAIL, violation of constraint TEST_CHECK1 caused operation
to fail
SQL> ROLLBACK;
```

CREATE TABLE Statement

```
SQL> -- This table shows that a CHECK column constraint
SQL> -- can refer to other tables in column select expressions:
SQL> CREATE TABLE TEST0
cont> ( TEST1 CHAR(5),
cont>   TEST2 CHAR(5)
cont>   CHECK (TEST2 NOT IN
cont>   (SELECT TEST1 FROM TEST0) )
cont> );
```

- An alternative to specifying unique column or table constraints is to use CREATE INDEX statements with the UNIQUE keyword. Specifying UNIQUE indexes generally gives better performance than specifying logically equivalent constraints in a table definition.
- The REFERENCES clause can declare the one or more corresponding columns in the referenced table that comprise a unique or primary key. If not, the referenced table must include a PRIMARY KEY constraint at the table level specifying the corresponding column or columns.
- If the dialect is SQL99 or ORACLE LEVEL2 then the list of columns in the REFERENCES clause need not match the order of the corresponding PRIMARY KEY or UNIQUE constraint. All other dialects require them to match.
- The values in a foreign key must match the values in the related unique key or primary key. SQL considers that the foreign key matches the related unique key or primary key when either of the following statements is true:
 - A column in the foreign key contains a null value. In this case, the foreign key is null. SQL considers that a null foreign key matches the related unique key or primary key.
 - None of the columns in the foreign key contains a null value, and the set of values in the foreign key also exists in the unique key or primary key. In other words, the foreign key matches the related unique key or primary key when for every row in the referencing table, there is a row in the referenced table where the corresponding columns are equal.

The following example illustrates the first type of match. The null value stored in column B2 of the table FOREIGN makes the foreign key of B1 and B2 a null foreign key. As a null foreign key, B1 and B2 match the primary key A1 and A2 in the table PRIMARY.

CREATE TABLE Statement

```
SQL> CREATE TABLE PRIMARY_TAB
cont> ( A1 INTEGER,
cont>   A2 INTEGER,
cont>   PRIMARY KEY (A1, A2),
cont>   A3 INTEGER);
SQL>
SQL> INSERT INTO PRIMARY_TAB (A1, A2, A3)
cont>         VALUES (1, 1, 1);
1 row inserted
SQL>
SQL> CREATE TABLE FOREIGN_TAB
cont> ( B1 INTEGER,
cont>   B2 INTEGER,
cont>   FOREIGN KEY (B1, B2)
cont>     REFERENCES PRIMARY_TAB (A1, A2),
cont>   B3 CHAR(5));
SQL> -- The following command stores a null value in column B2:
SQL> INSERT INTO FOREIGN_TAB (B1, B3) VALUES (2, 'AAAAA');
1 row inserted
```

This example shows the second type of match. The values stored in columns D1 and D2 (the foreign key) of the table FOREIGN_2 exactly match the values stored in columns C1 and C2 (the primary key) of the table PRIMARY_2.

```
SQL> CREATE TABLE PRIMARY_2
cont> (C1 INTEGER,
cont>   C2 INTEGER,
cont>   PRIMARY KEY (C1, C2),
cont>   C3 INTEGER);
SQL>
SQL> INSERT INTO PRIMARY_2 (C1, C2, C3)
cont>         VALUES (5, 3, 2);
1 row inserted
SQL>
SQL> CREATE TABLE FOREIGN_2
cont> ( D1 INTEGER,
cont>   D2 INTEGER,
cont>   FOREIGN KEY (D1, D2)
cont>     REFERENCES PRIMARY_2 (C1, C2),
cont>   D3 CHAR(5));
SQL> --
SQL> INSERT INTO FOREIGN_2 (D1, D2, D3) VALUES (5, 3, 'BBBBB');
1 row inserted
```

- You can use table-specific constraints to:
 - Maintain referential integrity by establishing a clear, visible set of rules
 - Attach the desired integrity rules directly to the definition of a table

CREATE TABLE Statement

- Avoid defining multiple, seemingly independent constraints to accomplish the same task
- Constraints should not specify columns defined as segmented strings, as only the segmented string ID is referenced, not the actual segmented string.
- Within the table definition, constraints can apply to the values in specific rows of a table, to the entire contents of a table, or to states existing between multiple tables.
- Within the table definition, Oracle Rdb first defines new versions of columns. Then, SQL defines constraints and evaluates them. Therefore, if columns and constraints are defined within the same table definition, constraints can use any of the columns defined in this table before or after the constraint definition text.
- If table-specific constraints are declared with a CREATE TABLE statement and the definition of the generated constraint fails, the definition of the table also fails.
- The CREATE TABLE statement adds the table definition and any associated constraint definitions to the physical database.
If the database was attached with the PATHNAME argument, the definitions are stored in the repository, ensuring consistency between the database definitions and the repository definitions.
- To ensure that you do not define redundant, table-specific constraints, you should display all constraints and triggers for the affected table using the SHOW TABLE statement.
- If a constraint fails at commit time, the update operation must be manually rolled back.
- You can create up to 8191 tables. This value is an architectural limit restricted by the on-disk structure and includes system tables. When you exceed the maximum limit, Oracle Rdb issues an error message.
If you delete older tables, Oracle Rdb recycles their identifiers so that CREATE TABLE statements can succeed even after the maximum is reached.
- CREATE TABLE statements in programs must precede (in the source file) all other data definition language (DDL) statements that refer to the table.

CREATE TABLE Statement

- You can specify the national character data type by using the NCHAR, NATIONAL CHAR, NCHAR VARYING, or NATIONAL CHAR VARYING data types. The national character data type is defined by the database national character set when the database is created. See Section 2.3 for more information regarding national character data types.
- You can specify the length of the data type in characters or octets. By default, data types are specified in octets. By preceding the CREATE TABLE command with the SET CHARACTER LENGTH or SET DIALECT statement, you change the length to characters. For more information, see the SET CHARACTER LENGTH Statement and the SET DIALECT Statement, respectively.
- Because data in temporary tables is private to a session, you cannot use temporary tables in as many places as you use persistent base tables. In particular, note the following points when you use temporary tables:
 - You can truncate global temporary tables using the TRUNCATE TABLE statement. You cannot truncate local temporary tables.
 - Global and local temporary tables cannot contain data of the data type LIST OF BYTE VARYING.
 - You can define column and table constraints for global temporary tables, but not for local temporary tables. The columns in both global and local temporary tables can reference domain constraints.
Constraints on a global temporary table can only refer to another global temporary table. However, if the referenced target table specifies ON COMMIT DELETE ROWS, the source table must also specify ON COMMIT DELETE ROWS. This restriction does not apply when the referenced target table specifies ON COMMIT PRESERVE ROWS.
 - You can use triggers with global temporary tables only.
 - You cannot define indexes for global or local temporary tables.
 - Oracle Rdb does not journal changes to global or local temporary tables.
- The following are allowed with global or local temporary tables:
 - You can delete temporary tables using the DROP TABLE statement.
 - A view can refer to a temporary table.
 - You can use dbkeys with temporary tables.
 - You can grant and revoke privileges only using the ALL keyword.
 - You can write to a temporary table during a read-only transaction.

CREATE TABLE Statement

- Table 7–1 summarizes the actions you can take with temporary tables and when you can refer to temporary tables.

Table 7–1 Using Temporary Tables

Action	Types of Temporary Tables		
	Global	Local	Declared Local
Drop table	Yes	Yes	No
Alter table	Yes	No	No
Truncate table	Yes	No	No
Add constraints on table or column	Yes	No	No
Refer to table in constraint definition	Yes ²	Yes	No
Refer to domain constraints	Yes	Yes	Yes
Refer to table in storage map	Yes ³	Yes	No
Refer to table in view	Yes	Yes	No
Grant privileges on temporary table	Yes	Yes	No
Refer to table in outline	Yes	Yes	No ¹
Create indexes on table	No	No	No
Use dbkeys on table	Yes	Yes	Yes
Use triggers with table	Yes	No	No
Refer to table in COMMENT ON statement	Yes	Yes	No
Contain LIST OF BYTE VARYING data	No	No	No
Specify in RESERVING clause	Yes ⁴	Yes ⁴	No
Write to table during read-only transaction	Yes	Yes	Yes
Create in a read-only transaction	No	No	Yes
Refer to a table in a computed by column	Yes	Yes	No

¹You can refer to a declared local temporary table if it is defined inside a stored module.

²From a temporary table only.

³Only the ENABLE or DISABLE COMPRESSION attribute may be specified.

⁴Such references are ignored.

CREATE TABLE Statement

For information about declared local temporary tables, see the `DECLARE LOCAL TEMPORARY TABLE` Statement.

- Data for a temporary table is stored in virtual memory, not in a storage area. For journaling purposes, when changes are made to the data in a temporary table such as updates or deletes, recovery space is required to hold before images of deleted and updated rows. This recovery space also requires virtual memory and may result in having to increase Page File Quota (process quota, `PGFLQUO`) and Virtual Page Count (`SYSGEN` parameter, `VIRTUALPAGECNT`) on OpenVMS.

A recommended way to reduce memory usage when using temporary tables is to commit transactions which modify temporary table data as soon as possible. Upon commit the additional copies of data are released and available for reuse by Oracle Rdb. This eliminates extra copies of data and therefore reduces virtual memory usage.

See the *Oracle Rdb Guide to Database Design and Definition* for calculating memory usage for temporary tables.

- When a constraint is disabled, it is not evaluated by the `INSERT`, `UPDATE`, `DELETE`, or `TRUNCATE TABLE` statements.
- The `RMU Verify` command with the `Constraint` qualifier ignores any disabled constraints. The exception is when a constraint is explicitly named using the `CONSTRAINT` option.
- The following usage notes apply to `AUTOMATIC` columns:
 - When the column is omitted from an insert operation, a column default and an automatic column provide similar functions. However, there are distinctions, as follows:
 - * `AUTOMATIC` columns cannot be referenced during an insert operation, because they are read-only to applications.
 - * `AUTOMATIC` columns can be written during an update operation.
 - * When you use an `AUTOMATIC` column, you do not provide the data type for the column.
 - Note the following differences between using `COMPUTED BY` columns and `AUTOMATIC` columns:
 - * `COMPUTED BY` columns use no space in the row, `AUTOMATIC` columns do.

CREATE TABLE Statement

- * A **COMPUTED BY** column is evaluated when the row is fetched, such as when a **SELECT**, **UPDATE**, or **DELETE** statement references the column name. An **AUTOMATIC** column is evaluated during an **INSERT** or **UPDATE** statement. A calculated value is written to a column in the row, and the value returned by a **SELECT** statement is the stored column value.

For example, a column defined as **COMPUTED BY CURRENT_DATE** returns the date when the query is executed. A selected column that is **AUTOMATIC INSERT AS CURRENT_DATE** returns the date when the **INSERT** was performed, which might be different from the date when the query is executed.

- * Indexes and constraints can be defined for **AUTOMATIC** columns but not for **COMPUTED BY** columns.
- Note the following differences between using an **AUTOMATIC** column and a trigger on the table:
 - * In an insert operation, an **AFTER INSERT TRIGGER** trigger can provide **AUTOMATIC** column functionality. However, **AUTOMATIC** columns can help eliminate the overhead of a trigger and so simplify table management.
 - * Trigger actions cannot modify a row being updated, because this leads to a recursive trigger action. **AUTOMATIC UPDATE** columns are evaluated prior to the trigger and constraint execution.
- If the data written to the table with an **AUTOMATIC** column is incorrect, you can temporarily suspend the read-only attribute of the column by issuing the **SET FLAGS 'AUTO_OVERRIDE'** statement if you have the **DBADMIN** privilege on the database. Then, you can execute an update query to correct the incorrect data. See the **SET FLAGS Statement** for more information and an example.
- The following usage notes apply to **UNIQUE** constraints:
 - Oracle Rdb provides an **SQL:1999-compliant UNIQUE** constraint. This type of constraint excludes **NULL** columns from the **UNIQUE** comparison. This effectively allows sets of columns to be **UNIQUE** or **NULL**.

This type of constraint is created by default when the **SQL** dialect is set to **SQL89**, **MIA**, **ORACLE LEVEL1**, **ORACLE LEVEL2**, **SQL99**, or **SQL92**. The default dialect is **SQLV40**. Oracle Corporation recommends that you set the dialect to **SQL99** (or one of the listed dialects) before using the **CREATE TABLE** statement (or **ALTER TABLE** statement) to add **UNIQUE** constraints to tables.

CREATE TABLE Statement

Note

The UNIQUE semantics are used at run time under any selected dialect. That is, the table must be created under the listed dialects to have the new style of UNIQUE constraints enabled.

- The SQL standard UNIQUE constraint implementation, in addition to conforming to the SQL Database Language standard, also provides improved performance for single row insert operations. This is made possible by eliminating checks for NULL values from the selection expression and thus simplifying the optimization for unique checking.

Here is a comparison of the old and new optimizer strategies. In this example, a UNIQUE constraint ("UNIQUE_A") and index on column A are used to check for uniqueness during an INSERT statement. Note that the optimizer chooses a full range search of the index (for example, [0:0]):

```
~S: Constraint "UNIQUE_A" evaluated
Cross block of 2 entries
  Cross block entry 1
    Conjunct      Firstn Get      Retrieval by DBK of relation T_UNIQUE
  Cross block entry 2
    Conjunct      Aggregate-F2    Conjunct
    Index only retrieval of relation T_UNIQUE
    Index name    T_UNIQUE_INDEX_A [0:0]
```

With the simplified UNIQUE constraint ("UNIQUE_B"), the optimizer can use a direct lookup of the index (that is, [1:1]), which reduces the I/O to the index to perform the constraint evaluation:

```
~S: Constraint "UNIQUE_B" evaluated
Cross block of 2 entries
  Cross block entry 1
    Conjunct      Firstn Get      Retrieval by DBK of relation T_UNIQUE
  Cross block entry 2
    Conjunct      Aggregate-F2    Index only retrieval of relation T_UNIQUE
    Index name    T_UNIQUE_INDEX_B [1:1]
```

- In prior versions, the UNIQUE constraint restricted columns to a single NULL value. To retain this behavior, use the SET DIALECT 'SQLV40' statement before creating new tables or altering existing tables to add UNIQUE constraints.

CREATE TABLE Statement

- UNIQUE constraints created in previous versions of Oracle Rdb will perform as in previous versions. Interfaces such as RDO or the Oracle CDD/Repository will continue to define the older style UNIQUE constraint. Database EXPORT and IMPORT will retain the UNIQUE constraint characteristics as defined by the database administrator, regardless of the defined dialect setting.

Note

The RMU Extract command with the Item=Table qualifier does not distinguish between the old and new UNIQUE constraints in this release of Oracle Rdb. You must modify the generated SQL script to establish the appropriate dialect before using the script to create a database.

- Because this new style of UNIQUE constraints is a relaxation of the UNIQUE rules, it is possible to drop the old style UNIQUE constraint and redefine the constraint under the SQL99 or similar dialect.
Note that this meaning of UNIQUE (that is, excluding NULL from the uniqueness test) does not apply to the UNIQUE index. The UNIQUE index still does not allow duplicate entries for NULL. If a UNIQUE index is currently defined that assists the UNIQUE constraint optimization, then the database administrator may want to drop the index and make it a non-UNIQUE index so that multiple NULLs can be stored. The UNIQUE constraint still enforces the uniqueness of the data.
- You can use the SQL SHOW TABLE command to determine which type of UNIQUE constraint is in use. See Example 16 in the Examples section.
- As a side effect of this change to UNIQUE constraints, Oracle Rdb also recognizes a larger class of CHECK constraints as being uniqueness checks. The main benefit is that these constraints are no longer executed when a DELETE statement is executed for the table, because DELETE statements do not affect the uniqueness of the remaining rows. For example:

CREATE TABLE Statement

```
SQL> CREATE TABLE T_USER_UNIQUE_NEW
cont>     A INTEGER,
cont>     B INTEGER,
cont>     CONSTRAINT UNIQUE_AB_NEW
cont>         CHECK ((SELECT COUNT(*)
cont>                   FROM T_USER_UNIQUE_NEW T2
cont>                   WHERE T2.A = T_USER_UNIQUE_NEW.A and
cont>                   T2.B = T_USER_UNIQUE_NEW.B) <= 1)
cont>         NOT DEFERRABLE
cont>     );
```

In previous versions of Oracle Rdb, only equality with 1 was recognized as a uniqueness constraint. In this example a comparison of LESS THAN or EQUAL TO 1 also qualifies as a uniqueness constraint.

See the *Oracle Rdb Guide to Database Design and Definition* for calculating memory usage for temporary tables.

- Only columns of the type TINYINT, SMALLINT, INTEGER, or BIGINT can use the IDENTITY attribute. These types must default to or have a zero scale. Domains may be referenced if they have these types.
- The IDENTITY attribute implicitly creates a system sequence with the same name as the table in which it resides. This sequence can be modified using the ALTER SEQUENCE statement, however, the sequence can only be dropped using ALTER TABLE . . . DROP COLUMN, or by DROP TABLE. There can only be one column using IDENTITY in any one table.
- The IDENTITY attribute implicitly changes the column to be an AUTOMATIC INSERT column, therefore it becomes a read-only column. Refer to the documentation on AUTOMATIC columns for more information.
- If a TRUNCATE TABLE is executed for a table with an IDENTITY column, the special sequence is reset to the initial starting value.
- DEFAULT and IDENTITY may not both be specified for a column.
- AUTOMATIC and IDENTITY may not both be specified for a column.
- Constraints, especially PRIMARY KEY, can be defined for the identity column.
- Indices can be defined which include the identity column.
- The table name provided by the LIKE clause must be a base table, a global temporary table, or a local temporary table that currently exists in the current database. You can also specify a synonym for a base table or temporary table.

CREATE TABLE Statement

The following attributes of the table are copied:

- The names and ordering of all columns
- For each column the data type, DEFAULT, IDENTITY, COMPUTED BY clause, AUTOMATIC AS clause, COMMENT and domain will be inherited.
- Display attributes such as DEFAULT VALUE, QUERY NAME, QUERY HEADER and EDIT STRING clauses.
- The table comment is inherited, unless overwritten by a COMMENT IS clause.
- If the source table includes an IDENTITY column then the LIKE clause will result in a new sequence to be created with the same name of this new table.

Other table attributes such as referential constraints, triggers, storage maps and indices are not inherited and must be separately created.

Note

If a COMPUTED BY expression uses a subselect to reference the current table then this information is inherited unchanged by the new table. You should perform a subsequent ALTER TABLE statement to DROP and redefine the COMPUTED BY column.

- You can not reference a system table or a view with the LIKE clause

```
SQL> create table my_sys like rdb$database;  
%RDB-E-NO_META_UPDATE, metadata update failed  
-RDMS-E-NOMETSYSREL, operation illegal on system defined metadata
```

- Any table referenced by a COMPUTED BY, AUTOMATIC or DEFAULT clause will be implicitly reserved for SHARED READ by Rdb when the column is referenced in a query. Therefore, it is not necessary to explicitly reserve these tables in the DECLARE TRANSACTION or SET TRANSACTION statement unless the required lock mode is higher than SHARED READ.

If any of these expressions call an SQL function which reads from a table or view, then these tables are not implicitly reserved. You must include a LOCK TABLE statement in the function (or any called procedure) to ensure that references to the tables are allowed, even when not listed

CREATE TABLE Statement

in the DECLARE TRANSACTION or SET TRANSACTION statement RESERVING clause.

Examples

Example 1: Creating new tables with primary and foreign keys

In this example, the CREATE TABLE statement is used to create the EMPLOYEES_2, SALARY_HISTORY_2, and WORK_STATUS_2 tables in the personnel database. It specifies column definitions based on domain definitions for the entire database.

The FOREIGN KEY constraint specified in the SALARY_HISTORY_2 table must match the PRIMARY KEY constraint specified in the EMPLOYEES_2 table.

Note also that the CHECK constraint specified is a table constraint because it is separated by commas from the column to which it refers. In this case, a column constraint on EMPLOYEE_ID would have the same effect because it refers only to the single column EMPLOYEE_ID.

Because the dialect is SQL99, the default for constraint evaluation time is NOT DEFERRABLE.

```
SQL> -- *** Set Dialect ***
SQL> --
SQL> SET DIALECT 'SQL99';
SQL> --
SQL> -- *** Create tables ***
SQL> --
SQL> CREATE TABLE WORK_STATUS_2
cont>  (
cont>  STATUS_CODE      STATUS_CODE_DOM
cont>  CONSTRAINT WS2_STATUS_CODE_PRIMARY
cont>  PRIMARY KEY,
cont>  STATUS_NAME      STATUS_NAME_DOM,
cont>  STATUS_TYPE      STATUS_DESC_DOM
cont>  );
SQL> --
SQL> CREATE TABLE EMPLOYEES_2
cont>  (
cont>  EMPLOYEE_ID      ID_DOM
cont>  CONSTRAINT E2_EMPLOYEE_ID_PRIMARY
cont>  PRIMARY KEY,
cont>  LAST_NAME        LAST_NAME_DOM,
cont>  FIRST_NAME       FIRST_NAME_DOM,
cont>  MIDDLE_INITIAL   MIDDLE_INITIAL_DOM,
cont>  ADDRESS_DATA_1   ADDRESS_DATA_1_DOM,
cont>  ADDRESS_DATA_2   ADDRESS_DATA_2_DOM,
cont>  CITY             CITY_DOM,
```

CREATE TABLE Statement

```
cont> STATE STATE_DOM,
cont> POSTAL_CODE POSTAL_CODE_DOM,
cont> SEX SEX_DOM
cont> CONSTRAINT EMPLOYEE_SEX_VALUES
cont> CHECK (
cont> SEX IN ('M', 'F') OR SEX IS NULL
cont> ),
cont> BIRTHDAY DATE_DOM,
cont> STATUS_CODE STATUS_CODE_DOM
cont> CONSTRAINT E2_STATUS_CODE_FOREIGN
cont> REFERENCES WORK_STATUS_2 (STATUS_CODE),
cont> CONSTRAINT EMP_STATUS_CODE_VALUES_2
cont> CHECK (
cont> STATUS_CODE IN ('0', '1', '2')
cont> OR STATUS_CODE IS NULL
cont> );
SQL> --
SQL> CREATE TABLE SALARY_HISTORY_2
cont> (
cont> EMPLOYEE_ID ID_DOM
cont> CONSTRAINT SH2_EMPLOYEES_ID_FOREIGN
cont> REFERENCES EMPLOYEES_2 (EMPLOYEE_ID),
cont> SALARY_AMOUNT SALARY_DOM,
cont> SALARY_START DATE_DOM,
cont> SALARY_END DATE_DOM
cont> );
SQL>
```

Example 2: Creating a table with many SQL data types

The following example is an excerpt from the sample program `sql_all_datatypes` created during installation of Oracle Rdb in the Samples directory. For a variety of languages, `sql_all_datatypes` illustrates how you declare program variables to match a variety of data types, and how you can specify those variables in SQL statements when you store and retrieve column values or null values.

This example shows the `CREATE TABLE` statement from the `sql_all_datatypes` program.

```
EXEC SQL CREATE TABLE ALL_DATATYPES_TABLE
(
CHAR_COL CHAR(10),
SMALLINT_COL SMALLINT,
SMALLINT_SCALED_COL SMALLINT(3),
INTEGER_COL INTEGER,
INTEGER_SCALED_COL INTEGER(2),
```

CREATE TABLE Statement

```
QUADWORD_COL          QUADWORD,
QUADWORD_SCALED_COL   QUADWORD (5),
REAL_COL              REAL,
DOUBLE_PREC_COL       DOUBLE PRECISION,
DATE_COL              DATE,
VARCHAR_COL           VARCHAR(40)
);
```

Example 3: Specifying default values for columns

The following example illustrates the use of default values for columns. Each salesperson enters his or her own daily sales information into the DAILY_SALES table.

```
SQL> --
SQL> CREATE TABLE DAILY_SALES
cont> --
cont> -- The column SALESPERSON is based on LAST_NAME_DOM and
cont> -- the default value is the user name of the person who
cont> -- enters the information:
cont> (SALESPERSON LAST_NAME_DOM DEFAULT USER,
cont> --
cont> -- Typical work day is 8 hours:
cont>   HOURS_WORKED SMALLINT DEFAULT 8,
cont>   HOURS_OVERTIME SMALLINT,
cont>   GROSS_SALES INTEGER );
SQL> --
SQL> -- Insert daily sales information accepting the
SQL> -- default values for SALESPERSON and HOURS_WORKED:
SQL> --
SQL> INSERT INTO DAILY_SALES
cont> (HOURS_OVERTIME, GROSS_SALES )
cont> VALUES
cont> (1, 2499.00);
1 row inserted
SQL> SELECT * FROM DAILY_SALES;
  SALESPERSON      HOURS_WORKED  HOURS_OVERTIME  GROSS_SALES
  KILPATRICK              8                1            2499
1 row selected
```

Example 4: Violating a constraint indirectly with the DELETE statement

Constraints prevent INSERT statements from adding rows to a table that do not satisfy conditions specified in the constraint. Constraints also prevent DELETE or UPDATE statements from deleting or changing values in a table if the deletion or change violates the constraint on another table in the database. The following example illustrates that point:

CREATE TABLE Statement

```
SQL> -- TEST has no constraints defined for it, but it is subject to
SQL> -- restrictions nonetheless because of the constraint specified
SQL> -- in TEST2:
SQL> CREATE TABLE TEST
cont> (COL1 REAL);
SQL>
SQL> CREATE TABLE TEST2
cont> (COL1 REAL,
cont> CHECK (COL1 IN
cont> (SELECT COL1 FROM TEST))
cont> );
SQL> COMMIT;
SQL>
SQL> INSERT INTO TEST VALUES (1);
1 row inserted
SQL> INSERT INTO TEST2 VALUES (1);
1 row inserted
SQL> COMMIT;
SQL> -- This DELETE statement will fail because it will cause COL1 in
SQL> -- TEST2 to contain a value without the same value in COL1 of TEST:
SQL> DELETE FROM TEST WHERE COL1 = 1;
1 row deleted
SQL> COMMIT;
%RDB-E-INTEG_FAIL, violation of constraint TEST2_CHECK1 caused operation to
fail
```

Example 5: Evaluating constraints at verb time

Deferrable constraints are not evaluated until a transaction issues a COMMIT statement. You can specify that constraints be evaluated more frequently with the EVALUATING clause of the SET TRANSACTION statement.

CREATE TABLE Statement

```
SQL> create table TEST
cont>     (col1 integer,
cont>     col2 integer
cont>     constraint C2
cont>     unique
cont>     deferrable
cont>     );
SQL>
SQL> insert into TEST (col1, col2) values (1, 2);
1 row inserted
SQL> commit;
SQL>
SQL> /*
***> This INSERT will violate the constraint as shown by
***> the error during COMMIT
***> */
SQL> insert into TEST (col1, col2) values (1, 2);
1 row inserted
SQL> commit;
%RDB-E-INTEG_FAIL, violation of constraint C2 caused operation to fail
-RDB-F-ON_DB, on database USER_DISK:[DOC.DATABASES]MF_PERSONNEL.RDB;1
SQL> /*
***> The COMMIT failed, so we will ROLLBACK
***> */
SQL> rollback;
SQL>
SQL> /*
***> You can change the evaluation time using the EVALUATING
***> clause of SET TRANSACTION
***> */
SQL> set transaction read write evaluating C2 at verb time;
SQL> insert into TEST (col1, col2) values (1, 2);
%RDB-E-INTEG_FAIL, violation of constraint C2 caused operation to fail
-RDB-F-ON_DB, on database USER_DISK:[DOC.DATABASES]MF_PERSONNEL.RDB;1
SQL> rollback;
```

Example 6: Specifying the DECIMAL data type in the CREATE TABLE statement

SQL does not support a packed decimal or numeric string data type. If you specify the DECIMAL or NUMERIC data type for a column in a CREATE TABLE or ALTER TABLE statement, SQL generates a warning message and creates the column with a data type that depends on the precision argument specified (see Section 2.3.3 for details). This example shows a CREATE TABLE statement that specifies a DECIMAL data type.

```
SQL> CREATE TABLE TEMP
cont>     (DECIMAL_EX DECIMAL);
%SQL-I-NO_DECIMAL, DECIMAL_EX is being converted from DECIMAL to INTEGER.
SQL>
```

CREATE TABLE Statement

Example 7: Basing a table on a repository record definition

In the following example, the FROM clause is used in a CREATE TABLE statement to create a table with constraints based on a repository record definition. The PARTS record (table) has a primary key based on the field (column) PART_ID and a unique key based on the field (column) PART_NO, as well as other constraints.

This example assumes that OTHER_PARTS record and OTHER_PARTS_ID field have been previously defined in the repository. It begins with defining the fields and the record in the repository using the Common Dictionary Operator utility.

```
$ !
$ ! Define CDD$DEFAULT:
$ !
$ DEFINE CDD$DEFAULT SYS$COMMON:[REPOSITORY]TABLE_TEST
$ !
$ ! Enter the repository to create new field and record definitions:
$ !
$ REPOSITORY
CDO> !
CDO> ! Create the field definitions for the PARTS record:
CDO> !
CDO> DEFINE FIELD PART_NO DATATYPE IS SIGNED WORD.
CDO> DEFINE FIELD PART_ID DATATYPE IS SIGNED LONGWORD.
CDO> DEFINE FIELD PART_ID_USED_IN DATATYPE IS SIGNED LONGWORD.
CDO> DEFINE FIELD PART_QUANT DATATYPE IS SIGNED WORD.
CDO> !
CDO> ! Create the PARTS record definition by first defining the constraints
CDO> ! and then including the field definitions just created. Note that
CDO> ! CDO creates the constraints as not deferrable.
CDO> !
CDO> DEFINE RECORD PARTS
cont>     CONSTRAINT PARTS_PMK PRIMARY KEY PART_ID
cont>     CONSTRAINT PARTS_UNQ UNIQUE PART_NO
cont>     CONSTRAINT PART_CST CHECK
cont>         (ANY P IN PARTS WITH (PART_ID IN
cont>             PARTS = PART_ID_USED_IN IN P))
cont>     CONSTRAINT PART_FRK
cont>         FOREIGN KEY PART_ID REFERENCES OTHER_PARTS OTHER_PART_ID.
cont>     PART_NO.
cont>     PART_ID.
cont>     PART_ID_USED_IN.
cont>     PART_QUANT.
cont>     END.
```

CREATE TABLE Statement

```
CDO> !
CDO> ! Display the RECORD PARTS:
CDO> !
CDO> SHOW RECORD PARTS/FULL
Definition of record PARTS
| Contains field          PART_NO
|   Datatype             signed word
| Contains field          PART_ID
|   Datatype             signed longword
| Contains field          PART_ID_USED_IN
|   Datatype             signed longword
| Contains field          PART_QUANT
|   Datatype             signed word
| Constraint PARTS_PMK   primary key PART_ID NOT DEFERRABLE
| Constraint PARTS_UNQ   unique PART_NO NOT DEFERRABLE
| Constraint PART_CST   (ANY (P IN PARTS WITH
|   (PART_ID IN PARTS EQ PART_ID_USED_IN IN P))) NOT DEFERRABLE
| Constraint PART_FRK   foreign key PART_ID references OTHER_PARTS
|   OTHER_PART_ID NOT DEFERRABLE
CDO> EXIT
$ !
$ ! Entering SQL:
$ SQL
SQL> !
SQL> ! Attach to the AUTO database:
SQL> !
SQL> ATTACH 'ALIAS AUTO PATHNAME AUTO';
SQL> !
SQL> ! Create a table called PARTS using the PARTS record (table)
SQL> ! just created in the repository:
SQL> !
SQL> CREATE TABLE FROM SYS$COMMON:[REPOSITORY]TABLE_TEST.PARTS
cont>     ALIAS AUTO;
SQL> !
SQL> ! Use the SHOW TABLE statement to display the information about the
SQL> ! PARTS table:
SQL> !
SQL> SHOW TABLE AUTO.PARTS;
Information for table AUTO.PARTS

CDD Pathname: SYS$COMMON:[REPOSITORY]TABLE_TEST.PARTS;1

Columns for table AUTO.PARTS:
Column Name  Data Type  Domain
-----
PART_NO      SMALLINT  AUTO.PART_NO
PART_ID      INTEGER   AUTO.PART_ID
PART_ID_USED_IN  INTEGER   AUTO.PART_ID_USED_IN
PART_QUANT   SMALLINT  AUTO.PART_QUANT
```

CREATE TABLE Statement

Table constraints for AUTO.PARTS:

AUTO.PARTS_PMK
Primary Key constraint
Table constraint for AUTO.PARTS
Evaluated on each VERB
Source: primary key PART_ID

AUTO.PARTS_UNQ
Unique constraint
Table constraint for AUTO.PARTS
Evaluated on each VERB
Source: unique PART_NO

AUTO.PART_CST
Check constraint
Table constraint for AUTO.PARTS
Evaluated on each VERB
Source: (ANY (P IN PARTS WITH (PART_ID IN PARTS EQ PART_ID_USED_IN IN P)))

AUTO.PART_FRK
Foreign Key constraint
Table constraint for AUTO.PARTS
Evaluated on each VERB
Source: foreign key PART_ID references OTHER_PARTS OTHER_PART_ID

Constraints referencing table AUTO.PARTS:
No constraints found

```
.  
.  
.  
SQL> --  
SQL> COMMIT;  
SQL> DISCONNECT DEFAULT;  
SQL> EXIT;
```

Example 8: Defining table-specific constraints with single-column primary and foreign keys

This example uses single-column keys to define table-specific constraints. The example maintains referential integrity among the four tables involved by using primary and foreign keys.

Three single-column primary key constraints preserve the integrity among the tables. The primary key constraints are the EMPLOYEE_ID column for the EMPLOYEES_TEST table, the JOB_CODE column for the JOBS_TEST table, and the DEPARTMENT_CODE column for the DEPARTMENTS_TEST table. The JOB_HISTORY_TEST table contains three foreign key constraints that refer to these primary keys.

CREATE TABLE Statement

Because the dialect is set to SQL99, constraints are NOT DEFERRABLE.

```
SQL> SET DIALECT 'SQL99';
SQL> --
SQL> CREATE TABLE EMPLOYEES_TEST
cont>   (EMPLOYEE_ID      ID_DOM
cont>       CONSTRAINT E_TEST_EMP_ID_PRIMARY
cont>       PRIMARY KEY,
cont>   LAST_NAME          LAST_NAME_DOM,
cont>   FIRST_NAME         FIRST_NAME_DOM,
cont>   MIDDLE_INITIAL     MIDDLE_INITIAL_DOM,
cont>   ADDRESS_DATA_1    ADDRESS_DATA_1_DOM,
cont>   ADDRESS_DATA_2    ADDRESS_DATA_2_DOM,
cont>   CITY               CITY_DOM,
cont>   STATE              STATE_DOM,
cont>   POSTAL_CODE        POSTAL_CODE_DOM,
cont>   SEX                SEX_DOM,
cont>   BIRTHDAY           DATE_DOM,
cont>   STATUS_CODE        STATUS_CODE_DOM);
SQL> --
SQL> CREATE TABLE JOBS_TEST
cont>   (JOB_CODE          JOB_CODE_DOM,
cont>       CONSTRAINT J_TEST_CODE_PRIMARY
cont>       PRIMARY KEY (JOB_CODE),
cont>   WAGE_CLASS         WAGE_CLASS_DOM,
cont>   JOB_TITLE          JOB_TITLE_DOM,
cont>   MINIMUM_SALARY    SALARY_DOM,
cont>   MAXIMUM_SALARY    SALARY_DOM);
SQL> --
SQL> CREATE TABLE DEPARTMENTS_TEST
cont>   (DEPARTMENT_CODE   DEPARTMENT_CODE_DOM,
cont>       CONSTRAINT D_DEPT_CODE_PRIMARY
cont>       PRIMARY KEY (DEPARTMENT_CODE),
cont>   DEPARTMENT_NAME    DEPARTMENT_NAME_DOM,
cont>   MANAGER_ID         ID_DOM,
cont>   BUDGET_PROJECTED   BUDGET_DOM,
cont>   BUDGET_ACTUAL      BUDGET_DOM);
SQL> --
```

CREATE TABLE Statement

```
SQL> CREATE TABLE JOB_HISTORY_TEST
cont>     (EMPLOYEE_ID      ID_DOM
cont>         CONSTRAINT JH_TEST_EMP_ID_FOREIGN
cont>         REFERENCES EMPLOYEES_TEST (EMPLOYEE_ID),
cont>     JOB_CODE           JOB_CODE_DOM
cont>         CONSTRAINT JH_J_CODE_FOREIGN
cont>         REFERENCES JOBS_TEST (JOB_CODE),
cont>     JOB_START          DATE_DOM,
cont>     JOB_END            DATE_DOM,
cont>     DEPARTMENT_CODE    DEPARTMENT_CODE_DOM
cont>         CONSTRAINT JH_D_CODE_FOREIGN
cont>         REFERENCES DEPARTMENTS_TEST (DEPARTMENT_CODE),
cont>     SUPERVISOR_ID      ID_DOM);
SQL>
```

Example 9: Defining table-specific constraints with multicolumn primary and foreign keys

The following example uses multicolumn keys to define table-specific constraints using a segment of the personnel database. This example uses some definitions not supplied with the sample database.

In this example, the two columns LOC and DEPT constitute a key, and they are defined as a **PRIMARY KEY** constraint for the WORK_STATION table. The two columns LOCATION and DEPARTMENT in the WORKER table are a foreign key that references the primary key in the WORK_STATION table.

Because the dialect is set to **SQL99**, constraints are **NOT DEFERRABLE**, and you do not receive a deprecated feature message when you define a constraint.

```
SQL> SET DIALECT 'SQL99';
SQL> --
SQL> CREATE DOMAIN LOC_DOM CHAR (10);
SQL> CREATE DOMAIN DEPT_DOM CHAR (10);
SQL> CREATE DOMAIN MGR_DOM CHAR (20);
SQL> CREATE DOMAIN NAME_DOM CHAR (20);
SQL> --
```

CREATE TABLE Statement

```
SQL> CREATE TABLE WORK_STATION
cont>   (LOC          LOC_DOM,
cont>   DEPT          DEPT_DOM,
cont>   CONSTRAINT WS_LOC_DEPT_PRIMARY
cont>   PRIMARY KEY (LOC, DEPT),
cont>   MGR            MGR_DOM);
SQL> --
SQL> CREATE TABLE WORKER
cont>   (NAME          NAME_DOM
cont>   CONSTRAINT WORKER_PRIMARY_NAME
cont>   PRIMARY KEY,
cont>   LOCATION        LOC_DOM,
cont>   DEPARTMENT      DEPT_DOM,
cont>   CONSTRAINT WORKER_FOREIGN_LOCATION_DEPT
cont>   FOREIGN KEY (LOCATION, DEPARTMENT)
cont>   REFERENCES WORK_STATION (LOC, DEPT));
SQL>
```

Example 10: Defining a table that contains a list

The following example defines a column of the data type LIST OF BYTE VARYING for storing employee resumes. This example defines the column EMPLOYEE_ID in the table EMPLOYEES as a foreign key constraint because resumes are kept only for actual employees for use in human resource management applications. Applications could use this table to identify employees with special backgrounds and skills for possible job assignments or promotions.

```
SQL> CREATE DOMAIN RESUME_DOM LIST OF BYTE VARYING;
SQL> CREATE TABLE RESUMES
cont>   (EMPLOYEE_ID    ID_DOM
cont>   REFERENCES EMPLOYEES (EMPLOYEE_ID),
cont>   RESUME           RESUME_DOM);
SQL> SHOW TABLE RESUMES;
Information for table RESUMES

Columns for table RESUMES:

Columns for table RESUMES:
Column Name          Data Type          Domain
-----
EMPLOYEE_ID          CHAR(5)            ID_DOM
Foreign Key constraint RESUMES_FOREIGN1
Unique constraint RESUMES_UNIQUE_EMPLOYEE_ID
RESUME               VARBYTE LIST      RESUME_DOM
Segment Length: 1
```

CREATE TABLE Statement

```
Table constraints for RESUMES:
RESUMES_FOREIGN1
  Foreign Key constraint
  Column constraint for RESUMES.EMPLOYEE_ID
  Evaluated on COMMIT
  Source:
      RESUMES.EMPLOYEE_ID REFERENCES EMPLOYEES (EMPLOYEE_ID)

RESUMES_UNIQUE_EMPLOYEE_ID
  Unique constraint
  Column constraint for RESUMES.EMPLOYEE_ID
  Evaluated on COMMIT
  Source:
      RESUMES.EMPLOYEE_ID UNIQUE

Constraints referencing table RESUMES:
No constraints found

Indexes on table RESUMES:
No indexes found

Storage Map for table RESUMES:
  RESUMES_MAP

Triggers on table RESUMES:
No triggers found

SQL>
```

Example 11: Defining a table with a computed column that uses a select expression

You can use a select expression in a **COMPUTED BY** clause. The following example shows how to use the **COMPUTED BY** clause to count the number of current employees of a particular department.

```
SQL> CREATE TABLE DEPTS1
cont>     (DEPARTMENT_CODE DEPARTMENT_CODE_DOM,
cont>     DEPT_COUNT COMPUTED BY
cont>     (SELECT COUNT (*) FROM JOB_HISTORY JH
cont>     WHERE JOB_END IS NULL
cont>     AND
cont>     --
cont>     -- Use correlation names to qualify the DEPARTMENT_CODE columns.
cont>     DEPTS1.DEPARTMENT_CODE = JH.DEPARTMENT_CODE),
cont>     DEPARTMENT_NAME DEPARTMENT_NAME_DOM)
cont> ;
SQL> SELECT * FROM DEPTS1 WHERE DEPARTMENT_CODE = 'ADMN';
DEPARTMENT_CODE  DEPT_COUNT  DEPARTMENT_NAME
ADMN              7           Corporate Administration
1 row selected
```


CREATE TABLE Statement

Example 12: Creating a table using the database default character set, national character set, and other character sets to define the columns

Assume the database was created defining the database default character set as DEC_KANJI and the national character set as KANJI.

```
SQL> CREATE TABLE COLOURS
cont>      (ENGLISH   MCS_DOM,
cont>      FRENCH     MCS_DOM,
cont>      JAPANESE   KANJI_DOM,
cont>      ROMAJI     DEC_KANJI_DOM,
cont>      KATAKANA   KATAKANA_DOM,
cont>      HINDI      HINDI_DOM,
cont>      GREEK      GREEK_DOM,
cont>      ARABIC     ARABIC_DOM,
cont>      RUSSIAN    RUSSIAN_DOM);
SQL> SHOW TABLE (COLUMNS) COLOURS;
Information for table COLOURS
```

Columns for table COLOURS:

Column Name	Data Type	Domain
ENGLISH	CHAR(8)	MCS_DOM
DEC_MCS 8 Characters,	8 Octets	
FRENCH	CHAR(8)	MCS_DOM
DEC_MCS 8 Characters,	8 Octets	
JAPANESE	CHAR(8)	KANJI_DOM
KANJI 4 Characters,	8 Octets	
ROMAJI	CHAR(16)	DEC_KANJI_DOM
KATAKANA	CHAR(8)	KATAKANA_DOM
KATAKANA 8 Characters,	8 Octets	
HINDI	CHAR(8)	HINDI_DOM
DEVANAGARI 8 Characters,	8 Octets	
GREEK	CHAR(8)	GREEK_DOM
ISOLATINGREEK 8 Characters,	8 Octets	
ARABIC	CHAR(8)	ARABIC_DOM
ISOLATINARABIC 8 Characters,	8 Octets	
RUSSIAN	CHAR(8)	RUSSIAN_DOM
ISOLATINCYRILLIC 8 Characters,	8 Octets	

CREATE TABLE Statement

Example 13: Creating and using a global temporary table

Assume that you have a base table called `PAYROLL` that is populated with data and that you want to extract the current week's information to generate paychecks for the company. The following example shows how to create a global temporary table called `PAYCHECKS_GLOB` and populate it with data from the `PAYROLL` and `EMPLOYEES` base tables. Your application can now operate on the data in `PAYCHECKS_GLOB` to calculate deductions and net pay for each employee. This eliminates continuous queries to the base tables and reduces concurrency conflicts.

```
SQL> CREATE GLOBAL TEMPORARY TABLE PAYCHECKS_GLOB
cont>     (EMPLOYEE_ID ID_DOM,
cont>     LAST_NAME CHAR(14),
cont>     HOURS_WORKED INTEGER,
cont>     HOURLY_SAL INTEGER(2),
cont>     WEEKLY_PAY INTEGER(2))
cont>     ON COMMIT PRESERVE ROWS;
SQL> --
SQL> -- Insert data into the temporary tables from other existing tables.
SQL> INSERT INTO PAYCHECKS_GLOB
cont>     (EMPLOYEE_ID, LAST_NAME, HOURS_WORKED, HOURLY_SAL, WEEKLY_PAY)
cont>     SELECT P.EMPLOYEE_ID, E.LAST_NAME, P.HOURS_WORKED, P.HOURLY_SAL,
cont>           P.HOURS_WORKED * P.HOURLY_SAL
cont>           FROM EMPLOYEES E, PAYROLL P
cont>           WHERE E.EMPLOYEE_ID = P.EMPLOYEE_ID
cont>           AND P.WEEK_DATE = DATE '1995-08-01';
100 rows inserted
SQL> --
SQL> -- Display the data.
SQL> SELECT * FROM PAYCHECKS_GLOB LIMIT TO 2 ROWS;
EMPLOYEE_ID  LAST_NAME      HOURS_WORKED  HOURLY_SAL  WEEKLY_PAY
00165        Smith          40            30.50       1220.00
00166        Dietrich       40            36.00       1440.00
2 rows selected
SQL> -- Commit the data.
SQL> COMMIT;
SQL> --
SQL> -- Because the global temporary table was created with PRESERVE ROWS,
SQL> -- the data is preserved after you commit the transaction.
SQL> SELECT * FROM PAYCHECKS_GLOB LIMIT TO 2 ROWS;
EMPLOYEE_ID  LAST_NAME      HOURS_WORKED  HOURLY_SAL  WEEKLY_PAY
00165        Smith          40            30.50       1220.00
00166        Dietrich       40            36.00       1440.00
2 rows selected
```

CREATE TABLE Statement

Example 14: Enabling and Disabling Constraints While Creating a Table

The PRIMARY KEY constraint enforces uniqueness on column A already. This example disables the additional UNIQUE constraint, leaving it to document the restriction but avoiding the evaluation at run-time.

```
SQL> SET DIALECT 'SQL99';
SQL> CREATE TABLE TT
cont> (A INTEGER CONSTRAINT A1 UNIQUE,
cont> CONSTRAINT A2 UNIQUE (A),
cont> CONSTRAINT A3 PRIMARY KEY (A))
cont> ENABLE CONSTRAINT A1
cont> DISABLE CONSTRAINT A2;
```

Example 15: Using AUTOMATIC Columns

This example uses automatic columns to fill in column values during INSERT and UPDATE.

Suppose that you want to store the current time stamp of a transaction and supply a unique numeric value for an order number. In addition, when the row is updated (the order is altered), you want a new time stamp to be written to the LAST_UPDATED column. You could write an application to supply this information, but you could not guarantee the desired behavior. For instance, a user with access to the table might update the table with interactive SQL and forget to enter a new time stamp to the LAST_UPDATED column. If you use an AUTOMATIC column instead, it can be defined so that columns automatically receive data during an insert operation. The data is sorted like any other column, but the column is read-only.

```
SQL> CREATE TABLE ORDER_HEADER
cont> (ORDER_NUMBER  AUTOMATIC INSERT AS NEW_ORDER.NEXTVAL,
cont> ORDER_DATE     AUTOMATIC INSERT AS CURRENT_TIMESTAMP,
cont> LAST_UPDATED   AUTOMATIC UPDATE AS CURRENT_TIMESTAMP
cont>              DEFAULT NULL,
cont> CUSTOMER_NUMBER INTEGER,
cont> ORDER_TOTAL    MONEY CHECK (ORDER_TOTAL >= 0.0));
```

CREATE TABLE Statement

Example 16: SHOW TABLE Output for Old and New UNIQUE Constraints

```
SQL> -- This first example is a UNIQUE constraint created when
SQL> -- the default dialect is used (SQLV40). A new description
SQL> -- follows the "Unique constraint" text, explaining the
SQL> -- interpretation of null values.
SQL> SHOW TABLE (CONSTRAINT) T_UNIQUE
Information for table T_UNIQUE
Table constraints for T_UNIQUE:
T_UNIQUE_UNIQUE_B_A
Unique constraint
Null values are considered the same
Table constraint for T_UNIQUE
Evaluated on UPDATE, NOT DEFERRABLE
Source:
```

```
        UNIQUE (b,a)
```

```
      .
      .
      .
```

```
SQL> -- This second example is a UNIQUE constraint created
SQL> -- when the dialect was set to 'SQL92', and the description
SQL> -- here indicates that all null values are considered
SQL> -- distinct.
SQL> SHOW TABLE (CONSTRAINT) T_UNIQUE2;
Information for table T_UNIQUE2
Table constraints for T_UNIQUE2:
T_UNIQUE2_UNIQUE_B_A
Unique constraint
Null values are considered distinct
Table constraint for T_UNIQUE2
Evaluated on UPDATE, NOT DEFERRABLE
Source:        UNIQUE (b,a)
```

```
      .
      .
      .
```

Example 17: Using the IDENTITY attribute

This simplified order entry database uses IDENTITY on all tables to generate unique values for the table primary key field.

CREATE TABLE Statement

```
SQL> set dialect 'SQL99';
SQL> create domain MONEY as INTEGER (2);
SQL> create domain CUSTOMER_ID as INTEGER;
SQL> create domain PRODUCT_ID as INTEGER;
SQL> create domain ORDER_ID as INTEGER;
SQL> create domain LINE_NUMBER as INTEGER
cont>     check (VALUE > 0 and VALUE IS NOT NULL)
cont>     not deferrable;
SQL>
SQL> create table PRODUCTS
cont>     (product_id          PRODUCT_ID identity primary key,
cont>     product_name         char (100),
cont>     unit_price            MONEY,
cont>     unit_name             char (10)
cont>     );
SQL> create unique index PRODUCTS_IX on PRODUCTS (product_id);
SQL>
SQL> create table CUSTOMERS
cont>     (customer_id         CUSTOMER_ID identity (1,1) primary key,
cont>     customer_name        char (100)
cont>     );
SQL> create unique index CUSTOMERS_IX on CUSTOMERS (customer_id);
SQL>
SQL> create table ORDERS
cont>     (order_id            ORDER_ID identity (1000) primary key,
cont>     order_date            timestamp,
cont>     customer_id          CUSTOMER_ID references CUSTOMERS
cont>     );
SQL> create unique index ORDERS_IX on ORDERS (order_id);
SQL>
SQL> create table ORDER_LINES
cont>     (order_id            ORDER_ID references ORDERS,
cont>     line_number           LINE_NUMBER,
cont>     product_id           PRODUCT_ID references PRODUCTS,
cont>     quantity              integer,
cont>     discount              float
cont>     );
SQL> create unique index ORDER_LINES_IX on ORDER_LINES (order_id, line_number);
SQL>
SQL> show sequences
Sequences in database with filename SAMPLE
    CUSTOMERS                An identity column sequence.
    ORDERS                    An identity column sequence.
    PRODUCTS                  An identity column sequence.
SQL> show sequences ORDERS
    ORDERS
Sequence Id: 3
An identity column sequence.
Initial Value: 1000
Minimum Value: 1000
Maximum Value: (none)
Next Sequence Value: 1000
```

CREATE TABLE Statement

```
Increment by: 1
Cache Size: 20
No Order
No Cycle
No Randomize
Wait
Comment: column IDENTITY sequence
SQL> commit;
```

As can be seen in the example the **START WITH** value was explicitly set to 1000, but the **INCREMENT BY** value was defaulted to 1.

Example 18: Defaulting all attributes of IDENTITY sequence

```
SQL> set dialect 'sql99';
SQL> create table PRODUCTS
cont>   (product_id      PRODUCT_ID identity primary key,
cont>   ...);
SQL> show sequence PRODUCTS;
PRODUCTS
Sequence Id: 1
An identity column sequence.
Initial Value: 1
Minimum Value: 1
Maximum Value: (none)
Next Sequence Value: 1
Increment by: 1
Cache Size: 20
No Order
No Cycle
No Randomize
Wait
Comment:      column IDENTITY sequence
```

As can be seen in the example both the **START WITH** and **INCREMENT BY** values for the sequence have defaulted to 1.

CREATE TABLE Statement

Example 19: Show that the IDENTITY sequence is reserved and can not be dropped

```
SQL> drop sequence ORDERS;
%RDB-E-NO_META_UPDATE, metadata update failed
-RDMS-E-NOMETSYSREL, operation illegal on system defined metadata
-RDMS-E-SEQNOTDEL, sequence "ORDERS" has not been deleted
```

Example 20: Adding an identity column to an existing table

A new table will be used to record the EMPLOYEES details after they are retired from the company. An extra column RETIRED_DATE is added to record the date of the retirement and a new CHECK constraint is added to ensure that that employee is not listed in both the EMPLOYEES table and the new RETIRED_EMPLOYEES table.

```
SQL> set dialect 'sql99';
SQL>
SQL> create table RETIRED_EMPLOYEES
cont>     like EMPLOYEES
cont>     (retired_date DATE_DOM
cont>     ,primary key (EMPLOYEE_ID)
cont>     ,check (not exists
cont>             (select * from EMPLOYEES e
cont>              where e.employee_id = RETIRED_EMPLOYEES.employee_id))
cont>     initially deferred
cont>     );
SQL>
SQL> show table RETIRED_EMPLOYEES;
Information for table RETIRED_EMPLOYEES

Columns for table RETIRED_EMPLOYEES:
Column Name  Data Type  Domain
-----
EMPLOYEE_ID  CHAR(5)    ID_DOM
LAST_NAME    CHAR(14)   LAST_NAME_DOM
FIRST_NAME   CHAR(10)   FIRST_NAME_DOM
MIDDLE_INITIAL CHAR(1)    MIDDLE_INITIAL_DOM
ADDRESS_DATA_1 CHAR(25)   ADDRESS_DATA_1_DOM
ADDRESS_DATA_2 CHAR(20)   ADDRESS_DATA_2_DOM
CITY         CHAR(20)   CITY_DOM
STATE        CHAR(2)    STATE_DOM
POSTAL_CODE  CHAR(5)    POSTAL_CODE_DOM
SEX          CHAR(1)    SEX_DOM
BIRTHDAY     DATE VMS   DATE_DOM
STATUS_CODE  CHAR(1)    STATUS_CODE_DOM
RETIRED_DATE DATE VMS   DATE_DOM
```

CREATE TABLE Statement

```
Table constraints for RETIRED_EMPLOYEES:
RETIRED_EMPLOYEES_CHECK1
  Check constraint
  Table constraint for RETIRED_EMPLOYEES
  Evaluated on COMMIT
  Source:
  CHECK (not exists
    (select * from EMPLOYEES e
      where e.employee_id = RETIRED_EMPLOYEES.employee_id))
RETIRED_EMPLOYEES_PRIMARY1
  Primary Key constraint
  Table constraint for RETIRED_EMPLOYEES
  Evaluated on UPDATE, NOT DEFERRABLE
  Source:
  PRIMARY key (EMPLOYEE_ID)

Constraints referencing table RETIRED_EMPLOYEES:
No constraints found

Indexes on table RETIRED_EMPLOYEES:
No indexes found

Storage Map for table RETIRED_EMPLOYEES:
No Storage Map found

Triggers on table RETIRED_EMPLOYEES:
No triggers found

SQL>
```

CREATE TRIGGER Statement

Creates triggers for a specified table. A **trigger** defines the actions to occur before or after the table is updated (by a write operation such as an INSERT, DELETE, or UPDATE statement). The trigger is associated with a single table, takes effect at a specific time for a particular type of update, and causes one or more triggered actions to be performed. If the trigger specifies multiple actions, each action is performed in the order in which it appears within the trigger definition.

With triggers, you can define useful actions such as:

- Cascading deletes
Deleting a row from one table causes additional rows to be deleted from other tables that are related to the first table by key values.
- Cascading updates
Updating a row in one table causes additional rows to be updated in other tables that are related to the first table by key values. These updates are commonly limited to the key fields themselves.
- Summation updates
Updating a row from one table causes a value in a row of another table to be updated by being increased or decreased.
- Hidden deletes
Causing rows to be deleted from a table by moving them to a parallel table that is not otherwise used by the database.

Note

Combinations of table-specific constraints and appropriately defined triggers, by themselves, are not sufficient to guarantee that database integrity is preserved when the database is updated. If integrity is to be preserved, table-specific constraints and triggers must be used in conjunction with a common set of update procedures that ensure completely reproducible and consistent retrieval and update strategies.

The CREATE TRIGGER statement adds the trigger definition to the physical database.

CREATE TRIGGER Statement

A **triggered action** consists of an optional predicate and some triggered statements. If specified, the predicate must evaluate to true for the triggered statements in the action to execute. Each triggered statement is executed in the order in which it appears within the triggered action clause.

The triggered statement can be:

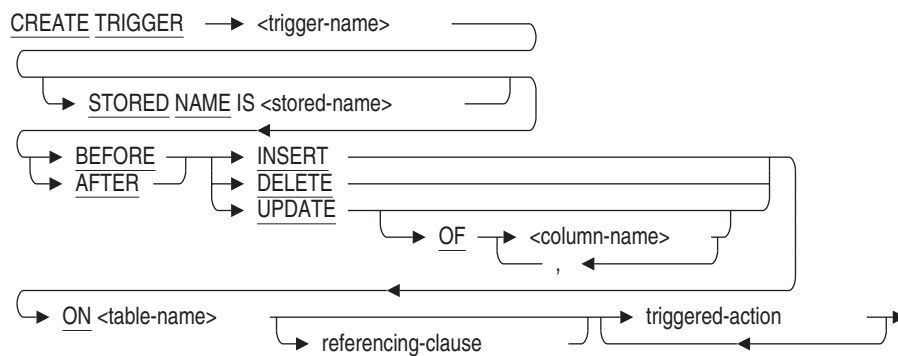
- A DELETE statement
- An UPDATE statement
- An INSERT statement
- A CALL statement
- A SIGNAL statement
- A TRACE statement
- An ERROR statement

Environment

You can use the CREATE TRIGGER statement:

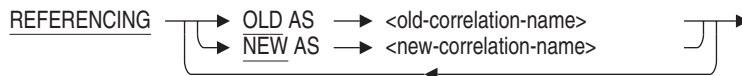
- In interactive SQL
- Embedded in host language programs to be precompiled
- As part of a procedure in an SQL module
- In dynamic SQL as a statement to be dynamically executed

Format

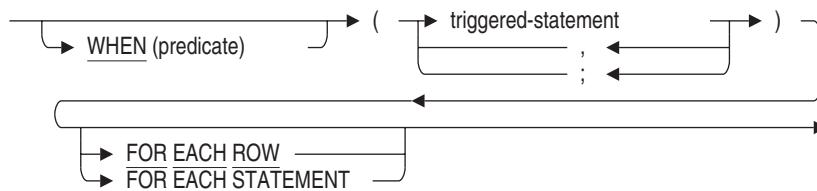


CREATE TRIGGER Statement

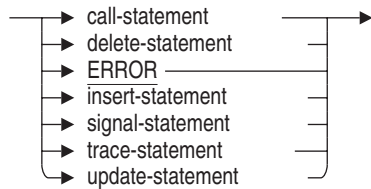
referencing-clause =



triggered-action =



triggered-statement =



Arguments

call-statement

Specifies the stored procedure to invoke. You can only call procedures with IN parameters. Operations on the triggering table are not permitted due to possible side effects and recursive calls.

column-name

The name of a column within the specified table to be checked for deletion, modification, or insertion. Use this argument only with UPDATE triggers.

delete-statement

Specifies the row of a table that you want to delete. If you specify CURRENT OF cursor-name with the WHERE clause of the DELETE statement, you receive an error message because the cursor is not visible to the CREATE TRIGGER statement.

CREATE TRIGGER Statement

ERROR

Provides the following message:

```
RDMS-E-TRIG_ERROR, Trigger 'trigger_name' forced an error.
```

A triggered ERROR statement cancels the DELETE, UPDATE, or INSERT statement that invoked the trigger.

FOR EACH ROW FOR EACH STATEMENT

Specifies whether the triggered action is evaluated once per triggering statement, or for each row of the subject table that is affected by the triggering statement.

If you specify FOR EACH STATEMENT, then the triggered action is evaluated only once, and row values are not available to the triggered action.

The FOR EACH STATEMENT clause is the default.

insert-statement

Specifies the new row or rows you want to add to a table.

old-correlation-name

A temporary name used to refer to the row values as they existed before an UPDATE operation occurred. If you do not specify the FOR EACH ROW clause, this correlation name cannot be referred to in the triggered statement.

new-correlation-name

A temporary name used to refer to the new row values to be applied by the UPDATE operation. If you do not specify the FOR EACH ROW clause, this correlation name cannot be referred to in the triggered statement.

referencing-clause

Lets you specify whether you want to refer to the row values as they existed before an UPDATE operation occurred or the new row values after they are applied by the UPDATE operation. Do not use this clause with INSERT or DELETE operations.

You can specify each option (OLD AS old-correlation-name or NEW AS new-correlation-name) only once in the referencing clause.

signal-statement

Specifies that the signaled SQLSTATE status parameter is to be passed back to the application or SQL interface and that the current routine and all calling routines are to be terminated. This provides a more complete error mechanism than is provided by the ERROR clause.

CREATE TRIGGER Statement

STORED NAME IS stored-name

Specifies a name that Oracle Rdb uses to access a trigger created in a multischema database. The stored name allows you to access multischema definitions using interfaces, such as Oracle RMU, the Oracle Rdb management utility, that do not recognize multiple schemas in one database. You cannot specify a stored name for a trigger in a database that does not allow multiple schemas. For more information on stored names, see Section 2.2.18.

table-name

The name of the table for which this trigger is defined.

trace-statement

Allows applications to add triggers to log information when trace logging is active.

triggered-action

Consists of an optional predicate, some triggered statements, and an optional frequency clause. If specified, the predicate must evaluate to true for the triggered statements in the triggered action clause to execute. Each triggered statement is executed in the order in which it appears within the triggered action clause.

triggered-statement

Updates the database or generates an error message.

update-statement

Specifies the row of a table that you want to modify. If you specify CURRENT OF cursor-name with the WHERE clause of the UPDATE statement, you receive an error message because the cursor is not visible to the CREATE TRIGGER statement.

WHEN (predicate)

Describes the optional condition that must be satisfied before the associated triggered statements are executed. This predicate cannot refer to any host language variable.

To avoid ambiguity between columns and external function callouts, use parentheses around the predicate in the WHEN clause. See the Usage Notes for further explanation.

CREATE TRIGGER Statement

Usage Notes

- If you did not attach to the database by a path name, the trigger definition is not stored in the repository. This causes an inconsistency between the definitions in the database and the repository. Therefore, you must define the triggers again whenever you restore the database metadata from the repository using the INTEGRATE statement.
- Creating a trigger requires SELECT and CREATE access to the subject table, and if any triggered statement specifies some form of update operation, also requires SELECT, DBCTRL, and the appropriate type of update (DELETE, UPDATE, INSERT) access to the tables specified by the triggered action statement.
- The trigger specification includes an action time, an update event (some type of write operation to the database), and an optional column list, which together determine when the trigger is to be evaluated. The action time can be specified as either before or after the update event (the INSERT, DELETE, or UPDATE statement). For triggers evaluated on UPDATE statements, you can specify an optional list of columns (from the subject table) to further stipulate that the trigger is to be evaluated only when one of the columns listed is also listed in the SET column list of the UPDATE statement. The trigger will be evaluated whether or not the values within the listed columns are actually changed during the execution of the UPDATE statement.
- Appropriate conditions may be placed in the WHEN predicate using both the NEW and OLD context values to prevent the execution of the trigger action if the actual column values did not change during the update.
- The frequency clause, FOR EACH ROW, determines whether an action is evaluated once per triggering statement, or for each row of the subject table that is affected by the triggering statement. If the FOR EACH ROW clause is not specified, the action is evaluated only once, and row values are not available to the triggered action.
- The table correlation name (current correlation name), old correlation name, and new correlation name are for various states of the subject table context of the triggered statement. The old correlation name is available (valid) only for AFTER UPDATE triggers and the new correlation name is available (valid) only for BEFORE UPDATE triggers.

CREATE TRIGGER Statement

- The trigger being defined checks for conflicts with the specified trigger for either update time and type, or in one of the column names on the list of columns to be modified. A triggered statement cannot affect the table on which the trigger is defined such that the trigger would be recursively invoked.
- Table 7–2 lists the six possible types of update action. Only one trigger specifying one of the six combinations of action time and type of update statement can be defined for any table. For update type UPDATE, this uniqueness is further qualified by any specified column names. A triggered statement cannot affect the table on which the trigger is defined such that the trigger would be recursively invoked.

The values from the row affected by the triggering statement are available to the triggered actions, as shown in Table 7–2.

Table 7–2 Availability of Row Data for Triggered Actions

Action Time/Type of Update	Availability of Row Data
BEFORE INSERT	Row data is not available.
AFTER INSERT	Row data referred to by the table correlation name is available.
BEFORE DELETE	Row data referred to by the table correlation name is available.
AFTER DELETE	Row data is not available.
BEFORE UPDATE	Old values of row data referred to by the table correlation name are available. New values of row data referred to by the new correlation name are available.
AFTER UPDATE	New values of row data referred to by the table correlation name are available. Old values of row data referred to by the old correlation name are available.

For example, a BEFORE INSERT trigger action for the EMPLOYEES table cannot create a row in the JOB_HISTORY table for the ID in the EMPLOYEE_ID column to be stored because the information in the row to be stored is not yet available. However, an AFTER INSERT trigger action can use the EMPLOYEE_ID column of the row being stored to create a row in the JOB_HISTORY table.

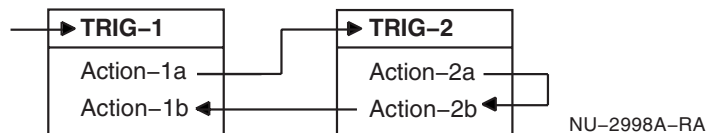
CREATE TRIGGER Statement

A BEFORE DELETE trigger action for the EMPLOYEES table can delete rows in the JOB_HISTORY table using the EMPLOYEE_ID column of the row to be deleted. However, an AFTER DELETE trigger action cannot delete any JOB_HISTORY rows using that EMPLOYEE_ID column because the information from the deleted row is no longer available.

- Once a trigger is selected for evaluation, SQL evaluates each pertinent triggered action in succession. The execution of a triggered action statement may cause other triggers to be selected for invocation; however, if a trigger is selected recursively by a direct or indirect execution of one of its actions, an exception is produced. Once all triggered actions have been exhausted, another pertinent trigger may be selected for evaluation (BEFORE UPDATE and AFTER UPDATE triggers only).
- An existing trigger cannot be changed. If you want to modify an existing trigger, you must delete it, then create a new trigger.
- The number in the third element of the SQLERRD array, SQLERRD[2], and the number displayed at the end of a statement in interactive SQL do not include the rows inserted, updated, and deleted by triggers.
- You must execute the CREATE TRIGGER statement in a read/write transaction. If you issue this statement when there is no active transaction, SQL starts a read/write transaction implicitly.
- Attempts to create a trigger fail if that trigger or its affected tables are involved in a query at the same time. Users must detach from the database with a DISCONNECT statement before you can create the trigger. When Oracle Rdb first accesses an object such as the table, a lock is placed on that object and not released until the user exits the database. If you attempt to update this object, you get a LOCK CONFLICT ON CLIENT message due to the other user's access to the object.
- You cannot execute the CREATE TRIGGER statement when the RDB\$SYSTEM storage area is set to read-only. You must first set RDB\$SYSTEM to read/write. See the *Oracle Rdb7 Guide to Database Performance and Tuning* for more information on the RDB\$SYSTEM storage area.
- Other users are allowed to be attached to the database when you issue the CREATE TRIGGER statement.
- If a trigger references a table not specified in the RESERVING clause of the SET TRANSACTION statement, that table is reserved as SHARED WRITE. If the table referenced by a trigger is already reserved in an incompatible mode, the statement that activates it fails.

CREATE TRIGGER Statement

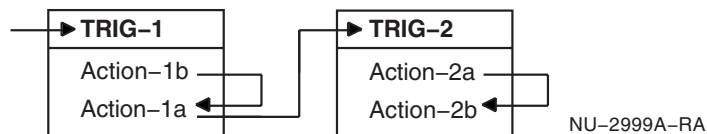
- If you invoke a trigger performing more than one action and one of those actions invokes another trigger, the actions performed in the second trigger must complete before the subsequent actions of the first trigger are executed. For example:



When TRIG-1 is invoked, Action-1a is executed which invokes TRIG-2. All actions of TRIG-2 must complete before any subsequent actions of TRIG-1 can execute. The actions of TRIG-1 and TRIG-2 occur in the following order:

Action-1a
Action-2a
Action-2b
Action-1b

The actions of TRIG-2 are not affected by the results of Action-1b because Action-1b does not execute until TRIG-2 is complete. Should you need the result of Action-1b to affect the results of TRIG-2, reverse the actions in TRIG-1. For example:



The actions of TRIG-1 and TRIG-2 now occur in the following order:

Action-1b
Action-1a
Action-2a
Action-2b

- The inclusion of a function call in a value expression causes ambiguity with conditional trigger definitions.

For example, the following syntax is ambiguous:

CREATE TRIGGER Statement

```
.  
. .  
WHEN '00190' <> EMPLOYEE_ID (ERROR)  
. .  
.
```

In the preceding example, it is difficult to determine if the predicate refers to the column `EMPLOYEE_ID` followed by an action or error, or if the predicate refers to a function call to the function `EMPLOYEE_ID` with an argument of `ERROR`. To support function calls within trigger definitions, SQL assumes this is a function call.

Use parentheses around the predicate in the `WHEN` clause to avoid this ambiguity.

- Oracle Rdb tracks language semantics for each trigger. If the language semantics are altered, the trigger is invalidated and must be re-created. The following semantics are fixed at data definition time:
 - `SELECT * FROM table-name`
The asterisk (*) expands to a column list
 - `INSERT INTO table-name VALUES (...)`
The column list defaults to the current names and order of the tables
 - Natural join
The matching names are used for equijoins

For example:

```
SQL> CREATE TRIGGER AFTER_T AFTER INSERT ON T  
cont> (INSERT INTO S VALUES (T.ID, T.SEQ)) FOR EACH ROW;  
SQL> ALTER TABLE S ADD COLUMN P REAL;  
%RDB-W-META_WARN, metadata successfully updated with the reported warning  
- RDMS-W-TRIG_LANGSEMEXI, table used by trigger with language dependency -  
trigger invalid on COMMIT  
SQL> COMMIT;  
. .  
.  
SQL> INSERT INTO T VALUES (0,0);  
%RDB-E-TRIG_REQ_ERROR, error encountered by a request using triggers  
- RDMS-E-TRG_INVALID, trigger can not be invoked - it is marked invalid  
-RDMS-E-TRIG_ERROR, trigger AFTER_T forced an error
```

CREATE TRIGGER Statement

- Oracle Rdb creates dependencies between triggers and other database objects, such as tables and routines, on which it depends. See Table 6-4 in the CREATE MODULE Statement which lists operations that may cause trigger invalidation.
- The SET FLAGS statement or the RDMS\$SET_FLAGS logical must be defined as TRACE to enable the TRACE statement prior to accessing the table that activates the trigger. Otherwise, the TRACE statement is not processed.
- When a SIGNAL statement is executed, the name of the TRIGGER is reported as the signalling routine.
- The CALL statement may activate SQL or external procedures as a trigger action with the following restrictions:
 - All parameters must be defined with mode IN because procedures may not update columns on the trigger table.
 - A SQL procedure must not execute an INSERT, DELETE, or UPDATE statement.
 - The SQL procedure may not use a CALL statement or activate a stored function in a value expression.
 - A transaction may not be started (SET TRANSACTION, START TRANSACTION, START DEFAULT TRANSACTION) or stopped (COMMIT, ROLLBACK) in the SQL procedure.

Only SQL procedures that use SELECT, procedural statements (such as CASE, WHILE, REPEAT, and FOR counted loops), subselects, or call external routines can be called from a trigger.

SQL procedures created by prior releases which conform to these restrictions may still be rejected by the CREATE TRIGGER process. The diagnostic would be similar to that shown in the following example. This occurs because the correct execution state was not recorded for this routine when it was created. This can be corrected by using the DROP and CREATE commands to create a new version.

An alternate method to correct the problem is to use the SET FLAGS VALIDATE_ROUTINE option as shown in the following example. Once the validation has been performed, a COMMIT command stores the state in the RDB\$ROUTINES system table for future use.

CREATE TRIGGER Statement

```
SQL> set transaction read write;
SQL>
SQL> -- attempt to use SQL procedure fails
SQL> create trigger T_A
cont>     after insert on M_TABLE
cont>     (call SEND_MAIL ('MANAGER', M_TABLE.last_name))
cont>     for each row;
%RDB-E-NO_META_UPDATE, metadata update failed
-RDB-E-RTN_FAIL, routine "SEND_MAIL" failed to compile or execute successfully
-RDMS-E-NOTRIGRTN, this stored routine may not be called from a trigger
SQL>
SQL> set flags 'VALIDATE_ROUTINE';
SQL> -- use NOEXECUTE so the routine is just compiled, not executed
SQL> set noexecute;
SQL> -- validate the routine to set the correct routine state
SQL> call SEND_MAIL ('MANAGER', '');
SQL> set execute;
SQL>
SQL> -- now the routine can be successfully used with the trigger definition
SQL> create trigger T_A
cont>     after insert on M_TABLE
cont>     (call SEND_MAIL ('MANAGER', M_TABLE.last_name))
cont>     for each row;
SQL>
SQL> commit;
```

- You can use a semicolon (;) as a separator between multiple trigger statements of a triggered action, for example:

```
SQL> create table AUDIT (su char(31), ct timestamp, a integer);
SQL> create table DATA_TABLE (a integer, b integer);
SQL>
SQL> create trigger T_AUDIT
cont>     after insert on DATA_TABLE
cont>     (trace 'before audit...';
cont>     insert into AUDIT (su, ct, a)
cont>     values (session_user, current_timestamp, DATA_TABLE.a);
cont>     trace 'after audit...')
cont>     for each row;
```

The use of a semicolon is required for the TRACE statement because comma (,) is an argument separator. SQL cannot distinguish the end of one statement from the next without a semicolon.

- If the CALL statement is used as a trigger action, or if a stored function is called from a trigger action (INSERT, DELETE, UPDATE, or CALL argument) or in the WHEN clause, the following restrictions apply:
 - The SQL function or procedure may not execute an INSERT, DELETE, or UPDATE statement.

CREATE TRIGGER Statement

- The SQL function or procedure may not use a CALL statement or activate another stored function in a value expression.

Oracle Corporation plans to relax these restrictions in a future release of Oracle Rdb.

Examples

Example 1: Defining a cascading delete trigger

The following SQL procedure shows a trigger from the sample personnel database that deletes rows in several tables before deleting a row in the EMPLOYEES table. Each associated employee row (from the tables that have foreign keys referring to the primary key in the employee row) is deleted. The employee identification number being deleted (00164) belongs to an employee who is also a manager; therefore, the MANAGER_ID column in the DEPARTMENTS table is set to null, as specified by the trigger.

```
SQL> SET TRANSACTION READ WRITE;
SQL> --
SQL> -- Display the EMPLOYEE_ID_CASCADE_DELETE trigger
SQL> -- in the sample database:
SQL> --
SQL> SHOW TRIGGER EMPLOYEE_ID_CASCADE_DELETE
      EMPLOYEE_ID_CASCADE_DELETE
Source:
EMPLOYEE_ID_CASCADE_DELETE
  BEFORE DELETE ON EMPLOYEES
  (DELETE FROM DEGREES D WHERE D.EMPLOYEE_ID =
   EMPLOYEES.EMPLOYEE_ID)
   FOR EACH ROW
  (DELETE FROM JOB_HISTORY JH WHERE JH.EMPLOYEE_ID =
   EMPLOYEES.EMPLOYEE_ID)
   FOR EACH ROW
  (DELETE FROM SALARY_HISTORY SH WHERE SH.EMPLOYEE_ID =
   EMPLOYEES.EMPLOYEE_ID)
   FOR EACH ROW
! Also, if an employee is terminated and that employee
! is the manager of a department, set the manager_id
! null for that department.
  (UPDATE DEPARTMENTS D SET D.MANAGER_ID = NULL
   WHERE D.MANAGER_ID = EMPLOYEES.EMPLOYEE_ID)
   FOR EACH ROW
```

CREATE TRIGGER Statement

```
SQL> --
SQL> -- The EMPLOYEES table has a value of '00164'
SQL> -- in the EMPLOYEE_ID column:
SQL> --
SQL> SELECT * FROM EMPLOYEES E WHERE E.EMPLOYEE_ID = '00164';
EMPLOYEE_ID  LAST_NAME          FIRST_NAME  MIDDLE_INITIAL
ADDRESS_DATA_1  ADDRESS_DATA_2    CITY
STATE  POSTAL_CODE  SEX  BIRTHDAY    STATUS_CODE
00164          Toliver      Alvin      A
146 Parnell Place
NH      03817      M      28-Mar-1947  1          Chocorua

1 row selected
SQL> --
SQL> --
SQL> -- The DEGREES table has two values of '00164'
SQL> -- in the EMPLOYEE_ID column:
SQL> --
SQL> SELECT * FROM DEGREES D WHERE D.EMPLOYEE_ID = '00164';
EMPLOYEE_ID  COLLEGE_CODE  YEAR_GIVEN  DEGREE  DEGREE_FIELD
00164        PRDU          1973       MA      Applied Math
00164        PRDU          1982       PhD     Statistics

2 rows selected
SQL> --
SQL> --
SQL> -- The JOB_HISTORY table has the value of '00164' in
SQL> -- several rows in the EMPLOYEE_ID column:
SQL> --
SQL> SELECT * FROM JOB_HISTORY JH WHERE JH.EMPLOYEE_ID = '00164';
EMPLOYEE_ID  JOB_CODE  JOB_START    JOB_END    DEPARTMENT_CODE
SUPERVISOR_ID
00164        DMGR      21-Sep-1981  NULL      MBMN
00228

00164        SPGM      5-Jul-1980   20-Sep-1981  MCBM
00164

2 rows selected
SQL> --
SQL> --
SQL> -- The SALARY_HISTORY table has a value of '00164'
SQL> -- in several rows in the EMPLOYEE_ID column:
SQL> --
SQL> SELECT * FROM SALARY_HISTORY SH WHERE SH.EMPLOYEE_ID = '00164';
EMPLOYEE_ID  SALARY_AMOUNT  SALARY_START  SALARY_END
00164        $26,291.00    5-Jul-1980    2-Mar-1981
00164        $51,712.00    14-Jan-1983    NULL
00164        $26,291.00    2-Mar-1981    21-Sep-1981
00164        $50,000.00    21-Sep-1981    14-Jan-1983

4 rows selected
SQL> --
```

CREATE TRIGGER Statement

```
SQL> --
SQL> -- The DEPARTMENTS table has a value of '00164'
SQL> -- in the MANAGER_ID column:
SQL> --
SQL> SELECT * FROM DEPARTMENTS D WHERE D.MANAGER_ID = '00164';
DEPARTMENT_CODE  DEPARTMENT_NAME      MANAGER_ID
BUDGET_PROJECTED  BUDGET_ACTUAL
MBMN              Board Manufacturing North      00164
                NULL              NULL
```

1 row selected

```
SQL> --
SQL> --
SQL> -- Test the trigger by deleting the row with a value of '00164'
SQL> -- in the EMPLOYEE_ID column from the EMPLOYEES table:
SQL> --
SQL> DELETE FROM EMPLOYEES E WHERE E.EMPLOYEE_ID = '00164';
1 row deleted
SQL> --
SQL> -- The row with a value of '00164' in the EMPLOYEE_ID column
SQL> -- was deleted from the EMPLOYEES table:
SQL> --
SQL> SELECT * FROM EMPLOYEES E WHERE E.EMPLOYEE_ID = '00164';
0 rows selected
SQL> --
SQL> -- The rows with a value of '00164' in the EMPLOYEE_ID column
SQL> -- were deleted from the DEGREES table:
SQL> --
SQL> SELECT * FROM DEGREES D WHERE D.EMPLOYEE_ID = '00164';
0 rows selected
SQL> --
SQL> -- The rows with a value of '00164' in the EMPLOYEE_ID
SQL> -- column were deleted from the JOB_HISTORY table:
SQL> --
SQL> SELECT * FROM JOB_HISTORY JH WHERE JH.EMPLOYEE_ID = '00164';
0 rows selected
SQL> --
SQL> -- The rows with a value of '00164' in the EMPLOYEE_ID
SQL> -- column were deleted from the SALARY_HISTORY table:
SQL> --
SQL> SELECT * FROM SALARY_HISTORY SH WHERE SH.EMPLOYEE_ID = '00164';
0 rows selected
SQL> --
SQL> -- The value of '00164' in the MANAGER_ID column was set to null
SQL> -- in the DEPARTMENTS table:
SQL> --
SQL> SELECT * FROM DEPARTMENTS D WHERE D.DEPARTMENT_CODE = 'MBMN';
DEPARTMENT_CODE  DEPARTMENT_NAME      MANAGER_ID
BUDGET_PROJECTED  BUDGET_ACTUAL
MBMN              Board Manufacturing North      NULL
                NULL              NULL
```

CREATE TRIGGER Statement

```
1 row selected
SQL> --
SQL> ROLLBACK;
```

Example 2: Defining a trigger that performs an update

Before the `STATUS_CODE` column in `WORK_STATUS` table is updated, the `STATUS_CODE_CASCADE_UPDATE` trigger in the following SQL procedure updates the associated rows in the `EMPLOYEES` table. The `REFERENCING` clause specifies `OLD_WORK_STATUS` as the correlation name for the values in the `WORK_STATUS` table before the `UPDATE` statement executes, and `NEW_WORK_STATUS` as the correlation name for the values in the `WORK_STATUS` table after the `UPDATE` statement executes.

```
SQL> -- Display the STATUS_CODE_CASCADE_UPDATE trigger in
SQL> -- the sample database:
SQL> --
SQL> SHOW TRIGGER STATUS_CODE_CASCADE_UPDATE
STATUS_CODE_CASCADE_UPDATE
Source:
STATUS_CODE_CASCADE_UPDATE
        BEFORE UPDATE OF STATUS_CODE ON WORK_STATUS
        REFERENCING OLD AS OLD_WORK_STATUS
                  NEW AS NEW_WORK_STATUS
        (UPDATE EMPLOYEES E
         SET E.STATUS_CODE = NEW_WORK_STATUS.STATUS_CODE
         WHERE E.STATUS_CODE = OLD_WORK_STATUS.STATUS_CODE)
        FOR EACH ROW

SQL> --
SQL> -- Change the STATUS_CODE column with a value of 2 to a value of 3:
SQL> --
SQL> UPDATE WORK_STATUS WS SET STATUS_CODE="3" WHERE STATUS_CODE="2";
1 row updated
SQL> --
SQL> -- The trigger changes any STATUS_CODE column in the EMPLOYEES table
SQL> -- with a value of 2 to a value of 3. Therefore, no rows are
SQL> -- selected for the first query that follows, but several are selected
SQL> -- for the second query:
SQL> --
SQL> SELECT * FROM EMPLOYEES E WHERE E.STATUS_CODE = "2";
0 rows selected
SQL> --
SQL> SELECT * FROM EMPLOYEES E WHERE E.STATUS_CODE = "3";
EMPLOYEE_ID  LAST_NAME          FIRST_NAME  MIDDLE_INITIAL
ADDRESS_DATA_1  ADDRESS_DATA_2    CITY
STATE  POSTAL_CODE  SEX  BIRTHDAY    STATUS_CODE
00165      Smith        Terry      D
120 Tenby Dr.
NH      03817      M      15-May-1954  3
Chocorua
```


CREATE TRIGGER Statement

```
00178      Goldstone      Neal      NULL
  194 Lyons Av,          Colebrook
    NH      03576      M      25-Apr-1952  3
.
.
.
00358      Lapointe      Jo Ann      C
  70 Tenby Dr.          Chocorua
    NH      03817      F      24-Feb-1931  3
```

12 rows selected

SQL> --

SQL> ROLLBACK;

Example 3: Defining a trigger that updates a sales summary

The following example defines a trigger that updates a monthly sales total after each daily sale is made.

```
SQL> --
SQL> -- Create the table to keep track of monthly sales:
SQL> CREATE TABLE MONTHLY_SALES
cont> ( SALES_AMOUNT INTEGER);
SQL> --
SQL> -- Create the table to keep track of sales made today:
SQL> CREATE TABLE DAILY_SALES
cont> ( SALES_AMOUNT INTEGER);
SQL> --
SQL> -- Assume that $250.00 of sales have been made during the current month:
SQL> INSERT INTO MONTHLY_SALES
cont> (SALES_AMOUNT) VALUES (250);
1 row inserted
SQL> --
SQL> -- After adding a new value to the SALES_AMOUNT column in
SQL> -- DAILY_SALES table, SQL updates the SALES column in
SQL> -- the MONTHLY_SALES table with the amount of the new sale:
SQL> CREATE TRIGGER UPDATE_SALES_TOTAL_ON_NEW_SALE
cont> AFTER INSERT ON DAILY_SALES
cont> (UPDATE MONTHLY_SALES M
cont>      SET M.SALES_AMOUNT = M.SALES_AMOUNT + DAILY_SALES.SALES_AMOUNT)
cont> FOR EACH ROW;
SQL> --
SQL> -- The following statement records a new $5.00 sale for today:
SQL> INSERT INTO DAILY_SALES
cont> (SALES_AMOUNT) VALUES (5);
1 row inserted
SQL> --
```

CREATE TRIGGER Statement

```
SQL> -- The value for the SALES_AMOUNT column of the DAILY_SALES table
SQL> -- is $5.00 and the value of the SALES_AMOUNT column of the
SQL> -- MONTHLY_SALES table is $255.00:
SQL> SELECT * FROM DAILY_SALES;
  SALES_AMOUNT
            5
1 row selected
SQL> --
SQL> SELECT * FROM MONTHLY_SALES;
  SALES_AMOUNT
            255
1 row selected
SQL> --
SQL> -- When a new $9.00 sale is made, the values in the two rows of the
SQL> -- SALES_AMOUNT column of the DAILY_SALES table are $5.00 and $9.00
SQL> -- and the value of the SALES_AMOUNT column of the MONTHLY_SALES
SQL> -- table is $264.00:
SQL> INSERT INTO DAILY_SALES
cont> (SALES_AMOUNT) VALUES (9);
1 row inserted
SQL> --
SQL> SELECT * FROM DAILY_SALES;
  SALES_AMOUNT
            5
            9
2 rows selected
SQL> --
SQL> SELECT * FROM MONTHLY_SALES;
  SALES_AMOUNT
            264
1 row selected
SQL> --
SQL> ROLLBACK;
SQL> --
```

Example 4: Defining a trigger that sets column values to null

Before the `STATUS_CODE` column in the `WORK_STATUS` table is deleted, this trigger causes the associated `WORK_STATUS` columns in the `EMPLOYEES` table to be set to null.

CREATE TRIGGER Statement

```
SQL> CREATE TRIGGER STATUS_CODE_ON_DELETE_SET_NULL
cont> BEFORE DELETE ON WORK_STATUS
cont> (UPDATE EMPLOYEES E SET E.STATUS_CODE = NULL
cont> WHERE E.STATUS_CODE = WORK_STATUS.STATUS_CODE)
cont> FOR EACH ROW;
SQL> --
SQL> -- Delete any row in the WORK_STATUS table where the STATUS_CODE
SQL> -- column has a value of 1:
SQL> DELETE FROM WORK_STATUS WS WHERE WS.STATUS_CODE = "1";
1 row deleted
SQL> --
SQL> -- This trigger sets the STATUS_CODE column value to null in many
SQL> -- rows in the EMPLOYEES table:
SQL> SELECT * FROM EMPLOYEES E WHERE E.STATUS_CODE IS NULL;
EMPLOYEE_ID LAST_NAME FIRST_NAME MIDDLE_INITIAL
ADDRESS_DATA_1 ADDRESS_DATA_2 CITY
STATE POSTAL_CODE SEX BIRTHDAY STATUS_CODE
00416 Ames Louie A
61 Broad st. NULL Alton
NH 03809 M 13-Apr-1941 NULL
00374 Andriola Leslie Q
111 Boston Post Rd. NULL Salisbury
NH 03268 M 19-Mar-1955 NULL
.
.
.
00200 Ziemke Al F
121 Putnam Hill Rd. NULL Winnisquam
NH 03289 M 27-Oct-1928 NULL

88 rows selected
SQL> ROLLBACK;
```

Example 5: Defining a trigger that prevents deletion of a row that exists in two tables

Suppose that a user wants to delete only those rows in the `JOB_HISTORY` table that do not also exist in the `JOBS` table. This is difficult to do with constraints because a row can exist in one table with a key number that does not exist in the other table. The following statement creates a trigger that causes an error when the user tries to delete a row that exists in table `JOB_HISTORY`.

CREATE TRIGGER Statement

```
SQL> CREATE TRIGGER DELETE_GUARD
cont> BEFORE DELETE ON JOB_HISTORY
cont> WHEN EXISTS (SELECT JOBS.JOB_CODE FROM JOBS
cont> WHERE JOBS.JOB_CODE=JOB_HISTORY.JOB_CODE)
cont> (ERROR) FOR EACH ROW;
SQL> --
SQL> -- Now attempt a deletion that violates the trigger.
SQL> --
SQL> DELETE FROM JOB_HISTORY WHERE JOB_CODE = 'DMGR';
%RDB-E-TRIG_INV_UPD, invalid update; encountered error condition
defined for trigger
-RDMS-E-TRIG_ERROR, trigger DELETE_GUARD forced an error
-RDB-F-ON_DB, on database DISK1:[DEPT3.SQL]MF_PERSONNEL.RDB;1
```

Example 6: Defining a trigger that saves audit information

```
SQL> -- Create new table to record changes made to
SQL> -- EMPLOYEES table
SQL> CREATE TABLE AUDIT_TRAIL
cont> (LOG DATE VMS,
cont> PERSON CHAR(31),
cont> TBL_NAME CHAR(10),
cont> OPER CHAR(1));
SQL> COMMIT;

SQL> -- Create a trigger so that each time
SQL> -- an INSERT operation is performed,
SQL> -- a record is stored in the AUDIT_TRAIL table.
SQL> CREATE TRIGGER EMPS_TRIGGER
cont> AFTER INSERT
cont> ON EMPLOYEES
cont> (INSERT INTO AUDIT_TRAIL
cont> VALUES (CURRENT_TIMESTAMP,
cont> CURRENT_USER, 'EMPLOYEES', 'I'))
cont> FOR EACH STATEMENT;
SQL> -- The AUDIT_TRAIL table currently has no records.
SQL> SELECT * FROM AUDIT_TRAIL;
0 rows selected
SQL> -- Insert a record into EMPLOYEES
SQL> INSERT INTO EMPLOYEES
cont> (EMPLOYEE_ID, LAST_NAME)
cont> VALUES ('00964', 'FRENCH');
1 row inserted
SQL> -- See if trigger updated the AUDIT_TRAIL table.
SQL> SELECT * FROM AUDIT_TRAIL;
LOG PERSON TBL_NAME OPER
17-JUN-2003 15:04:31.43 STEWART EMPLOYEES I
1 row selected
```

CREATE TRIGGER Statement

Example 7: Using TRACE as a trigger action

```
SQL> set flags 'TRACE';
SQL> create table M_TABLE (a integer, b integer);
SQL>
SQL> create trigger T_A
cont>     after insert on M_TABLE
cont>     (trace 'in a trigger: ' || cast(M_TABLE.a as varchar(10)))
cont>     for each row;
SQL>
SQL> insert into M_TABLE (a, b) values (1, 10);
~Xt: in a trigger: 1
1 row inserted
SQL>
```

Example 8: Using SIGNAL as a trigger action

```
SQL> create table M_TABLE (a integer, b integer);
SQL>
SQL> create trigger T_A
cont>     after insert on M_TABLE2
cont>     when (M_TABLE2.a is not null)
cont>     (signal '12345' ('in a trigger: '
cont>                               || cast(M_TABLE2.a as varchar(10))))
cont>     for each row;
SQL>
SQL> insert into M_TABLE2 (a, b) values (1, 10);
%RDB-E-SIGNAL_SQLSTATE, routine "T_A" signaled SQLSTATE "12345"
-RDB-I-TEXT, in a trigger: 1
SQL>
```

Example 9: Using CALL as a trigger action

```
SQL> create module M_MODULE
cont>     language SQL
cont>
cont>     procedure M_K (in :a int);
cont>         trace 'called from a trigger: ' || cast(:a as varchar(10));
cont>
cont> end module;
SQL>
SQL> create table M_TABLE (a integer, b integer);
SQL>
SQL> create trigger T_A
cont>     after insert on M_TABLE
cont>     (call M_K (M_TABLE.a))
cont>     for each row;
SQL>
SQL> insert into M_TABLE (a, b) values (1, 10);
~Xt: called from a trigger: 1
1 row inserted
SQL>
```

CREATE USER Statement

CREATE USER Statement

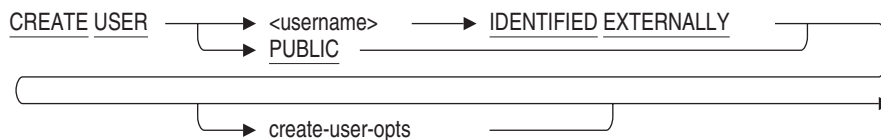
Creates a special security profile entry to identify a database user. That user can be granted roles, which in turn provide access to database objects.

Environment

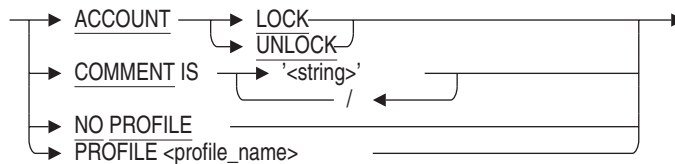
You can use the CREATE USER statement:

- In interactive SQL
- Embedded in host language programs
- As part of a procedure in an SQL module or other compound statement
- In dynamic SQL as a statement to be dynamically executed

Format



create-user-opts =



Arguments

ACCOUNT LOCK ACCOUNT UNLOCK

The ACCOUNT LOCK clause disables access to the database by the user for whom the CREATE USER statement is being applied. The ACCOUNT UNLOCK clause allows that user access to the database.

The ACCOUNT UNLOCK clause is the default.

CREATE USER Statement

COMMENT IS 'string'

Adds a comment about the user. SQL displays the text of the comment when it executes a SHOW USERS statement. Enclose the comment in single quotation marks (') and separate multiple lines in a comment with a slash mark (/).

IDENTIFIED EXTERNALLY

Indicates that the user will be authenticated through the operating system.

NOPROFILE

NOPROFILE is the default behavior and indicates that no special restrictions are applied to this user.

PROFILE

Identifies a new profile for assignment to the user. The specified profile name must be the name of an existing profile.

PUBLIC

Explicitly creates a PUBLIC security profile entry in the database.

username

The name of the user to add to the database. This must match the name of an existing OpenVMS username.

Usage Notes

- You must have the SECURITY privilege on the database to create a user.
- The special user PUBLIC exists implicitly. However, the CREATE USER statement can be used to create an explicit PUBLIC entry so that roles and profiles can be associated with the PUBLIC user. This allows control of anonymous users who access the database.
- You can display existing users defined for a database by issuing a SHOW USERS statement.
- The username must conform to OpenVMS naming conventions, that is, uppercase letters, numbers, underscore and '\$' with no spaces or punctuation.
- If SECURITY CHECKING IS INTERNAL, then the GRANT statement will implicitly perform a CREATE USER if the user is not defined in the database and the name exists as an OpenVMS user. The following example causes the user to be created.

CREATE USER Statement

```
SQL> grant ADMIN_USER to SMITH;
%RDB-W-META_WARN, metadata successfully updated with the reported warning
-RDMS-W-PRFCREATED, some users or roles were created
SQL> show users
Users in database with filename personnel
      SMITH
```

Examples

Example 1: Creating a New User and Locking Her Account

```
SQL> CREATE USER munroy IDENTIFIED EXTERNALLY
cont> ACCOUNT LOCK
cont> COMMENT IS 'User munroy starts job on'//
cont> 'May 1, 2003.  Unlock when she starts';
```

Example 2: Adding a profile to a user

This example creates a new profile that defines the DEFAULT transaction and then assigns a profile while creating a new user. The next time the user attaches to the database the START DEFAULT TRANSACTION statement will use the defined profile instead of the standard READ ONLY default.

```
SQL> create profile READ_COMMITTED
cont> default transaction read write isolation level read committed wait 30;
SQL> show profile READ_COMMITTED
      READ_COMMITTED
      Default transaction read write wait 30
      Isolation level read committed
SQL> create user JAIN identified externally profile READ_COMMITTED;
SQL> show user JAIN;
      JAIN
      Identified externally
      Account is unlocked
      Profile: READ_COMMITTED
      No roles have been granted to this user
```


CREATE VIEW Statement

Creates a view definition. A **view** is a logical structure that refers to rows stored in other tables. Data in a view is not physically stored in the database. You can include in a view definition combinations of rows and columns from other tables and view definitions in the schema. You define a view by specifying a select expression, that:

- Names the criteria for selecting the tables, rows, and columns for the view
- Specifies a set of columns from those tables

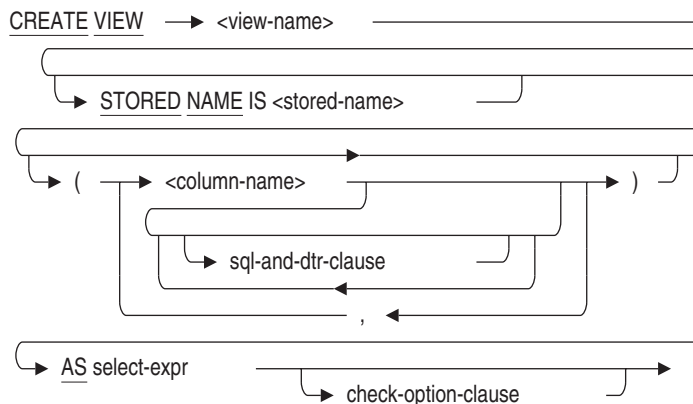
When the CREATE VIEW statement executes, SQL adds the view definition to the physical database. If you declared the schema with the PATHNAME argument, the definition is also stored in the repository.

Environment

You can use the CREATE VIEW statement:

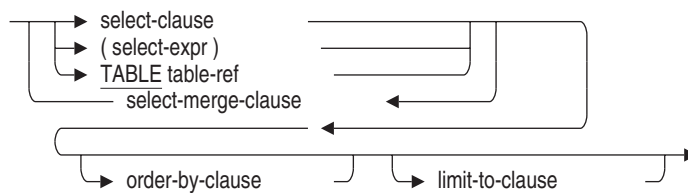
- In interactive SQL
- Embedded in host language programs to be precompiled
- As part of a procedure in an SQL module
- In dynamic SQL as a statement to be dynamically executed

Format

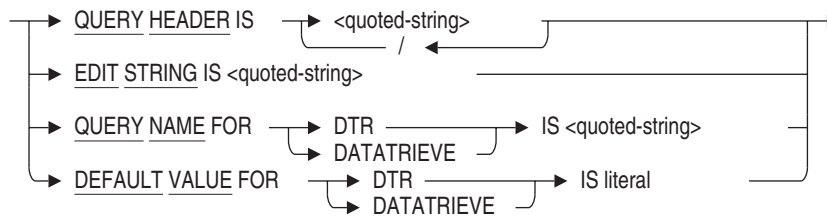


CREATE VIEW Statement

select-expr =



sql-and-dtr-clause =



check-option-clause =



Arguments

check-option-clause

A constraint that places restrictions on update operations made to a view. The check option clause ensures that any rows that are inserted or updated in a view conform to the definition of the view. Do not specify the WITH CHECK OPTION clause with views that are read-only. (The Usage Notes describe which views SQL considers read-only.)

column-name

A list of names for the columns of the view. If you omit column names, SQL assigns the names from the columns in the source tables in the select expression.

CREATE VIEW Statement

However, you must specify names for all the columns of the view in the following cases:

- The select expression generates columns with duplicate names.
- The select expression uses statistical functions or arithmetic expressions to create new columns that are not in the source tables.

CONSTRAINT check-option-name

Specify a name for the WITH CHECK OPTION constraint. If you omit the name, SQL creates a name. However, Oracle Rdb recommends that you always name constraints. If you supply a name for the WITH CHECK OPTION constraint, the name must be unique in the schema.

The name for the WITH CHECK OPTION constraint is used by the INTEG_FAIL error message when an INSERT or UPDATE statement violates the constraint.

select-expr

A select expression that defines which columns and rows of the specified tables SQL includes in the view. The select expression for a nonmultischema database can name only tables in the same schema as the view. A select expression for a multischema database can name a table in any schema in the database; the schema need not be in the same catalog as the view being created. See Section 2.8.1 for more information on select expressions.

sql-and-dtr-clause

Optional SQL and DATATRIEVE formatting clauses. See Section 2.5 for more information on formatting clauses.

STORED NAME IS stored-name

Specifies a name that Oracle Rdb uses to access a view created in a multischema database. The stored name allows you to access multischema definitions using interfaces, such as Oracle RMU, the Oracle Rdb management utility, that do not recognize multiple schemas in one database. You cannot specify a stored name for a view in a database that does not allow multiple schemas. For more details about stored names, see Section 2.2.18.

view-name

Name of the view definition you want to create. When choosing a name, follow these rules:

- Use a name that is unique among all view and table names in the schema.
- Use any valid SQL name (see Section 2.2 for more information).

CREATE VIEW Statement

Usage Notes

- Any statement that inserts, updates, or deletes rows of a view changes the rows of the base tables on which the view is based.
- Note the following when using INSERT, UPDATE, and DELETE statements that refer to views:
 - Do not refer to read-only views in INSERT, UPDATE, or DELETE statements. SQL considers as read-only views those with select expressions that:
 - * Use the DISTINCT argument to eliminate duplicate rows from the result table
 - * Name more than one table or view in the FROM clause
 - * Include a function in the select list
 - * Contain a UNION, EXCEPT (MINUS), INTERSECT, GROUP BY, or HAVING clause
- In INSERT and UPDATE statements, you cannot refer to columns in views that are the result of an arithmetic expression or a function. For instance, you cannot use an INSERT statement that refers to ARITH_COLUMN in the following view definition:

```
SQL> CREATE VIEW TEMP (ARITH_COLUMN, EMPLOYEE_ID)
cont> AS SELECT (SALARY_AMOUNT * 3), EMPLOYEE_ID
cont> FROM SALARY_HISTORY;
SQL>
SQL> INSERT INTO TEMP (ARITH_COLUMN) VALUES (111);
%RDB-E-READ_ONLY_FIELD, attempt to update read-only field ARITH_COLUMN
SQL> ROLLBACK;
```

- To allow correct SQLSTATE handling for the ANSI/ISO SQL standard, the exception raised by a WITH CHECK OPTION violation changes when the dialect is set to SQL99 at database attach time. For example:

```
SQL> SET DIALECT 'SQL99';
SQL> ATTACH 'FILENAME personnel_test';
SQL> INSERT INTO MANAGERS VALUES (1, 'Fred', 10);
%RDB-E-CHECK_FAIL, violation of view check option "MANAGERS_CHECKOPT1"
caused operation to fail
```

CREATE VIEW Statement

This change allows SQL to return a special SQLSTATE value of 44000 and allows applications to distinguish between constraint and view-check option violations. Adjust any error handlers that examine the RDB\$MESSAGE_VECTOR so that they correctly handle RDB\$_CHECK_FAIL (it is similar to the error RDB\$_INTEG_FAIL). For more information about SQLSTATE values, see Appendix C.

- Use the WITH CHECK OPTION clause to make sure that rows you insert or update in a view conform to its definition.

For example, the following view definition allows only salaries over \$60,000. Because you use the WITH CHECK OPTION clause, you cannot insert a row that contains a salary of less than \$60,000.

```
SQL> CREATE VIEW TEST
cont>   AS SELECT * FROM SALARY_HISTORY
cont>           WHERE SALARY_AMOUNT > 60000
cont>   WITH CHECK OPTION CONSTRAINT TEST_VIEW_CONST;
SQL>
SQL> INSERT INTO TEST (SALARY_AMOUNT) VALUES (50);
%RDB-E-INTEG_FAIL, violation of constraint TEST_VIEW_CONST-
caused operation to fail
```

- When you insert or update a view, the rows are stored in the base tables. If you do not use the WITH CHECK OPTION clause, you can insert or update rows through a view that do not conform to the view's definition. Once stored, however, you cannot retrieve those rows through the view because they do not meet the conditions specified by the view definition.

For instance, the following view definition allows only salaries over \$60,000. However, you can name the view in an INSERT statement to store a salary value of \$50, which you can then retrieve only by referring to the table on which the view is based.

CREATE VIEW Statement

```
SQL> CREATE VIEW TEMP
cont> AS SELECT * FROM SALARY_HISTORY
cont> WHERE SALARY_AMOUNT > 60000;
SQL>
SQL> INSERT INTO TEMP (SALARY_AMOUNT) VALUES (50);
1 row inserted
SQL> -- Cannot get the row just stored through the view TEMP:
SQL> --
SQL> SELECT * FROM TEMP WHERE SALARY_AMOUNT < 100;
0 rows inserted
SQL> -- To retrieve the row, select it from the base table
SQL> --
SQL> SELECT * FROM SALARY_HISTORY WHERE SALARY_AMOUNT < 100;
EMPLOYEE_ID SALARY_AMOUNT SALARY_START SALARY_END
NULL          50.00      NULL          NULL
1 row inserted
```

- You can create up to 53,247 views. These values are architectural limits restricted by the on-disk structure. When you exceed the maximum limit for views, Oracle Rdb issues the MAXVIEWID error message.

Views can have a record ID that ranges from 12288 through 65535.

If you delete older views, Oracle Rdb recycles their identifiers so that the CREATE VIEW statement can succeed even after reaching the maximum value.

Examples

Example 1: Defining a view based on a single table

This example shows a view definition that uses three columns from a single table, EMPLOYEES.

```
SQL> CREATE VIEW EMP_NAME
cont> AS SELECT
cont> FIRST_NAME,
cont> MIDDLE_INITIAL,
cont> LAST_NAME
cont> FROM EMPLOYEES;
SQL> --
SQL> -- Now display the rows from the view just created.
SQL> SELECT * FROM EMP_NAME;
FIRST_NAME MIDDLE_INITIAL LAST_NAME
Alvin      A              Toliver
Terry     D              Smith
.
.
.
```

Example 2: Defining a view that does not allow you to insert or update rows that do not conform to the view's definition

CREATE VIEW Statement

This example shows a view definition using the WITH CHECK OPTION clause.

```
SQL> CREATE VIEW ADMN_VIEW
cont> AS SELECT * FROM JOB_HISTORY
cont> WHERE DEPARTMENT_CODE = 'ADMN'
cont> WITH CHECK OPTION CONSTRAINT ADMN_VIEW_CONST;
SQL> -- You cannot insert a row that does not
SQL> -- conform to the view definition.
SQL> --
SQL> INSERT INTO ADMN_VIEW (DEPARTMENT_CODE) VALUES ('MBMN');
%RDB-E-INTEG-FAIL, violation of constraint ADMN_VIEW_CONST-
caused operation to fail
```

Example 3: Defining a view based on multiple tables

You can also define a view using more than one table.

```
SQL> CREATE VIEW CURRENT_SALARY
cont> AS SELECT
cont> E.LAST_NAME,
cont> E.FIRST_NAME,
cont> E.EMPLOYEE_ID,
cont> SH.SALARY_START,
cont> SH.SALARY_AMOUNT
cont> FROM
cont> SALARY_HISTORY SH, EMPLOYEES E
cont> WHERE
cont> SH.EMPLOYEE_ID = E.EMPLOYEE_ID
cont> AND
cont> SH.SALARY_END IS NULL ;
```

This example defines a view from the EMPLOYEES and SALARY_HISTORY tables. It uses the select expression to:

- Choose the columns derived from each table. Because no column names are specified before the select expression, the columns inherit the names from the source tables.
- Join the tables and limit the view to current salaries.

CREATE VIEW Statement

Example 4: Defining a view with local column names

```
SQL> CREATE VIEW EMP_JOB
cont>     ( CURRENT_ID,
cont>       CURRENT_NAME,
cont>       CURRENT_JOB,
cont>       SUPERVISOR )
cont> AS SELECT
cont>     E.EMPLOYEE_ID,
cont>     E.LAST_NAME,
cont>     J.JOB_TITLE,
cont>     JH.SUPERVISOR_ID
cont> FROM
cont>     EMPLOYEES E,
cont>     JOB_HISTORY JH,
cont>     JOBS J
cont> WHERE
cont>     E.EMPLOYEE_ID = JH.EMPLOYEE_ID
cont> AND
cont>     JH.JOB_CODE = J.JOB_CODE
cont> AND
cont>     JH.JOB_END IS NULL      ;
```

This view definition:

- Specifies local names for the columns in the view.
- Joins the EMPLOYEES and JOB_HISTORY tables. This join links rows in the EMPLOYEES table to rows in the JOB_HISTORY table.
- Joins the JOB_HISTORY and JOBS tables. This join lets the view contain job titles instead of job codes.
- Uses the JH.JOB_END IS NULL expression. This clause specifies that only the current JOB_HISTORY rows, where the JOB_END column is null, should be included in the view.

The following query uses the view defined in the previous example:

```
EXEC SQL
      DECLARE X CURSOR FOR
      SELECT CURRENT_ID, CURRENT_NAME, CURRENT_JOB, SUPERVISOR
      FROM EMP_JOB
END-EXEC

EXEC SQL
      OPEN X
END-EXEC

PERFORM WHILE SQLCODE NOT = 0
```


CREATE VIEW Statement

```
EXEC SQL
        FETCH X
        INTO :ID, :NAME, :JOB, :SUPER
END-EXEC

END PERFORM

EXEC SQL
        CLOSE X
END-EXEC
```

Example 5: Defining a view with a calculated column

This example shows a view definition that derives a column through a calculation based on a column in an base table.

```
SQL> CREATE VIEW SS_DEDUCTION
cont>     ( IDENT,
cont>         SALARY,
cont>         SS_AMOUNT )
cont> AS SELECT
cont>     E.EMPLOYEE_ID,
cont>     SH.SALARY_AMOUNT,
cont>     SH.SALARY_AMOUNT * 0.065
cont> FROM
cont>     SALARY_HISTORY SH, EMPLOYEES E
cont> WHERE
cont>     SH.EMPLOYEE_ID = E.EMPLOYEE_ID
cont> AND
cont>     SH.SALARY_END IS NULL ;
```

Each time the view column `SS_AMOUNT` is selected, it computes a new value from the `SALARY_AMOUNT` column of the `SALARY_HISTORY` table.

CREATE VIEW Statement

Example 6: Defining a view dependent on another view

This example creates a view, `DEPENDENT_VIEW`, that refers to the `CURRENT_JOB` view in its definition to include current job information for employees in the engineering department.

```
SQL> CREATE VIEW DEPENDENT_VIEW  
cont> AS SELECT * FROM CURRENT_JOB  
cont> WHERE DEPARTMENT_CODE = 'ENG';
```

DECLARE ALIAS Statement

Specifies the name and the source of the database definitions to be used for module compilation, and makes the named alias part of the implicit environment of an application. You can name either a file or a repository path name to be used for the database definitions.

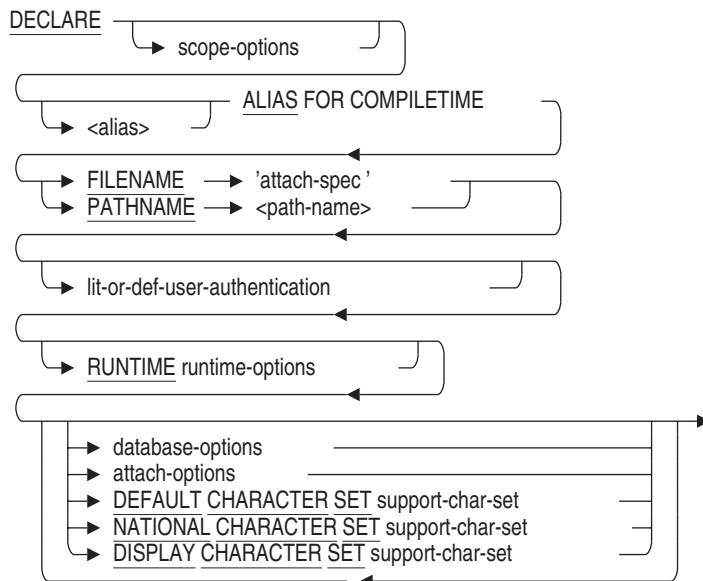
Environment

You can use the DECLARE ALIAS statement:

- Embedded in host language programs to be precompiled
- In a context file
- As part of the DECLARE section in an SQL module

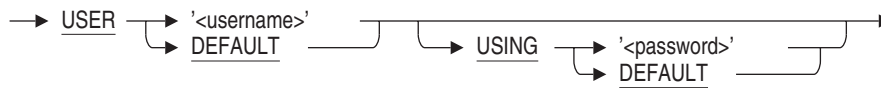
The alias that you declare must be different from any other alias specified in the module.

Format



DECLARE ALIAS Statement

lit-or-def-user-authentication =



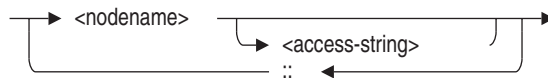
scope-options =



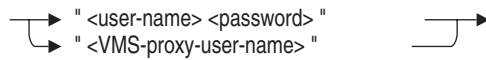
attach-spec =



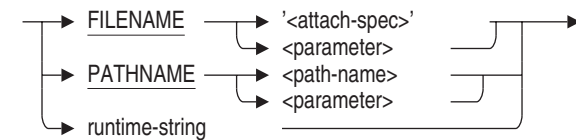
node-spec =



access-string =



runtime-options =

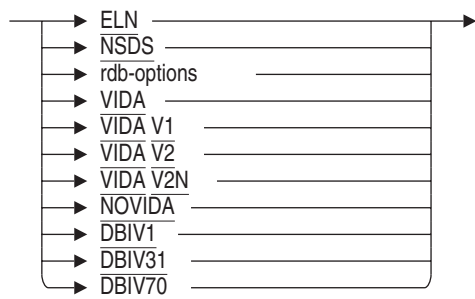


DECLARE ALIAS Statement

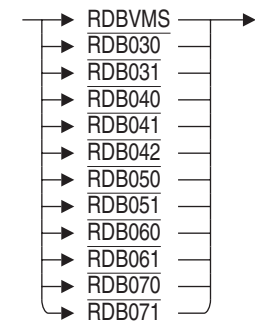
runtime-string =



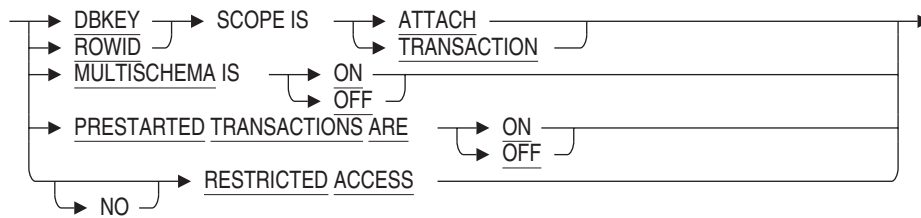
database-options =



rdb-options =



attach-options =



DECLARE ALIAS Statement

Arguments

alias ALIAS

Specifies a name for the attach to the database. Specifying an alias lets your program refer to more than one database.

You do not have to specify an alias in the DECLARE ALIAS statement. The default alias in interactive SQL and in precompiled programs is RDB\$DBHANDLE. In the SQL module language, the default is the alias specified in the module header. Using the default alias (either by specifying it explicitly in the DECLARE ALIAS statement or by omitting any alias) makes the database part of the default environment. Specifying a default database means that statements that refer to the default database do not need to use an alias.

If a default alias was already declared and you specify the default alias in the alias clause (or specify any alias that was already declared), you receive an error when you precompile the program or process it with the SQL module processor.

database-options

By default, SQL uses only the database options used to compile a program as valid options for that program. If you want to use the program with other supported databases, you can override the default options by specifying database options in the ATTACH or DECLARE ALIAS statement.

For more information on database options, see Section 2.10.

DBKEY SCOPE IS ATTACH DBKEY SCOPE IS TRANSACTION

Controls when the database key of a deleted row can be used again by SQL.

- The default DBKEY SCOPE IS TRANSACTION means that SQL can reuse the database key of a deleted table row (to refer to a newly inserted row) as soon as the transaction that deleted the original row completes with a COMMIT statement. (If the user who deleted the original row enters a ROLLBACK statement, then the database key for that row cannot be used again by SQL.)

During the connection of the user who entered the DECLARE ALIAS statement, the DBKEY SCOPE IS TRANSACTION clause specifies that a database key is guaranteed to refer to the same row *only* within a particular transaction.

DECLARE ALIAS Statement

- The DBKEY SCOPE IS ATTACH clause means that SQL cannot use the database key again (to refer to a newly inserted row) until all users who have attached with DBKEY SCOPE IS ATTACH have detached from the database.

It only requires one process to attach with DBKEY SCOPE IS ATTACH to force all database users to assume this characteristic.

- Oracle Corporation recommends using DBKEY SCOPE IS TRANSACTION to prevent excessive consumption of storage area space by overhead space needed to support DBKEY SCOPE IS ATTACH, and to prevent performance problems when storing new rows.

During the connection of the user who entered the DECLARE ALIAS statement, the DBKEY SCOPE IS ATTACH clause specifies that a database key is guaranteed to refer to the same row until the user detaches from the database.

See Section 2.6.5 for more information.

DEFAULT CHARACTER SET *support-char-set*

Specifies the default character set of the alias at compile time. For a list of allowable character set names, see Section 2.1.

DISPLAY CHARACTER SET *support-char-set*

Specifies the character set encoding and characteristics expected of text strings returned from Oracle Rdb. See the Usage Notes under CREATE DATABASE Statement for additional information.

FILENAME '*attach-spec*'

A quoted string containing full or partial information needed to access a database.

For an Oracle Rdb database, an attach specification contains the file specification of the .rdb file.

When you use the FILENAME argument, any changes you make to database definitions are entered *only* to the database system file, not to the repository. If you specify FILENAME, your application attaches to the database with that file name at run time.

If you specify FILENAME:

- During compilation, your application attaches to the specified database and reads metadata from the database definitions.
- At run time, your application attaches to the specified database.

For information regarding node-spec and file-spec, see Section 2.2.8.1.

DECLARE ALIAS Statement

FOR COMPILETIME

Optional keyword provided for upward compatibility: `DECLARE ALIAS` specifies the compile-time environment by default. Specifies that the alias declared is the source of the database definition for program compiling and execution.

lit-or-def-user-authentication

Specifies the user name and password to enable access to databases, particularly remote databases.

You can use this clause to explicitly provide user name and password information in the `DECLARE ALIAS` statement.

literal-user-auth

Specifies the user name and password for the specified database to be accessed at run time. For more information about when to use this clause, see the `ATTACH` Statement.

LOCAL

GLOBAL

EXTERNAL

Specifies the scope of the alias declaration in precompiled SQL or SQL module language.

The scope-option declarations are:

- `LOCAL` declares an alias that is local to procedures in the module in which it is declared, or local to dynamic statements prepared in the module in which it is declared.

SQL attaches to a database with `LOCAL` scope only when you execute a procedure in the same module without a session. The alias of a database with `LOCAL` scope pertains only to that module.

If the execution of a procedure in another module has attached to the implicit environment and that procedure subsequently calls another procedure that references a local database, SQL attempts to attach to that local database. If no transaction is active, SQL adds the local database to the implicit environment for this module. If a transaction is active, SQL returns an error message.

- `GLOBAL` declares an alias definition that is global to procedures in the application. `GLOBAL` is the default.
- `EXTERNAL` declares an external reference to a global alias that is defined in another module.

DECLARE ALIAS Statement

In single-image applications, the distinction between alias definitions and alias references is often unimportant. It is only necessary that each alias have at least one definition. For this reason, Oracle Rdb has treated all alias references (declared with the `EXTERNAL` keyword) the same as alias definitions (declared with the `GLOBAL` keyword or the default.) For compatibility with previous versions, this remains the default.

However, applications that share aliases between multiple images require a distinction between alias definitions and alias references. All definitions of any aliases shared between multiple OpenVMS images must be defined in one image, generally the shareable image against which you link the other images.

Oracle Rdb recommends that you distinguish alias definitions from alias references in any new source code. Use the `GLOBAL` (or default) scope keyword for alias definitions and the `EXTERNAL` keyword for alias references. If you share aliases between multiple OpenVMS images, use the `NOEXTERNAL_GLOBALS` command line qualifier to override the default and cause SQL to properly treat alias references as references.

If you use the `EXTERNAL_GLOBAL` command line qualifier, SQL treats aliases declared with the `EXTERNAL` keyword as `GLOBAL`. That is, SQL initializes alias references as well as alias definitions.

If you use the `NOEXTERNAL_GLOBAL` command line qualifier, SQL treats aliases declared with the `EXTERNAL` keyword as alias references and does not initialize them. It initializes all other aliases.

The `EXTERNAL_GLOBAL` qualifier is the default.

The `[NO]INITIALIZE_HANDLES` command line qualifiers also affect the initialization of aliases, but they are recommended only for use in versions prior to V7.0.

See Section 3.6 and Section 4.3 for more information about the command line qualifiers.

MULTISCHEMA IS ON MULTISCHEMA IS OFF

The `MULTISCHEMA IS ON` clause enables multischema naming for the duration of the database attach. The `MULTISCHEMA IS OFF` clause disables multischema naming for the duration of the database attach. Multischema naming is disabled by default.

NATIONAL CHARACTER SET support-char-set

Specifies the national character set of the alias at compile time. For a list of allowable character set names, see Section 2.1.

DECLARE ALIAS Statement

PATHNAME path-name

A full or relative repository path name that specifies the source of the database definitions. When you use the **PATHNAME** argument, any changes you make to database definitions are entered in both the repository and the database system file. Oracle Rdb recommends using the **PATHNAME** argument if you have the repository on your system and you plan to use any data definition statements.

If you specify **PATHNAME**:

- During compilation, your application attaches to the repository database definition and reads metadata from the dictionary definitions. SQL extracts the file name of the Oracle Rdb database from the dictionary and saves it for use at run time.
- At run time, your application attaches to the Oracle Rdb database file name extracted from the dictionary at compilation.

PRESTARTED TRANSACTIONS ARE ON PRESTARTED TRANSACTIONS ARE OFF

Specifies whether Oracle Rdb enables or disables prestarted transactions.

Use the **PRESTARTED TRANSACTIONS ARE OFF** clause only if your application uses a server process that is attached to the database for long periods of time and causes the snapshot file to grow excessively. If you use the **PRESTARTED TRANSACTIONS ARE OFF** clause, Oracle Rdb may require additional I/O as each **SET TRANSACTION** statement must reserve a transaction sequence number (TSN).

For most applications, Oracle Rdb recommends that you enable prestarted transactions. The default is **PRESTARTED TRANSACTIONS ARE ON**. If you use the **PRESTARTED TRANSACTIONS ARE ON** clause or do not specify the **PRESTARTED TRANSACTIONS** clause, the **COMMIT** or **ROLLBACK** statement for the previous read/write transaction automatically reserves the TSN for the next transaction and reduces I/O.

You can use **ALTER DATABASE . . . PRESTARTED TRANSACTIONS** clause to establish a default setting for all applications using the database. You can also define the **RDMS\$BIND_PRESTART_TXN** logical name to define the default setting for prestarted transactions outside of an application. The **PRESTARTED TRANSACTION** clause overrides this logical name and database setting. For more information, see the *Oracle Rdb7 Guide to Database Performance and Tuning*.

DECLARE ALIAS Statement

RESTRICTED ACCESS

NO RESTRICTED ACCESS

Restricts access to the database. This allows you to access the database but locks out all other users until you disconnect from the database. Setting restricted access to the database requires DBADM privileges.

The default is NO RESTRICTED ACCESS if not specified.

ROWID SCOPE IS ATTACH

ROWID SCOPE IS TRANSACTION

The ROWID keyword is a synonym for the DBKEY keyword. See the DBKEY SCOPE IS argument earlier in this Arguments list for more information.

RUNTIME runtime-options

Specifies the source of the database definitions when the program is run.

runtime-string

A quoted string or parameter that specifies the file name or path name of the database to be accessed at run time, and optionally, the user name and password of the user accessing the database at run time.

USER 'username'

USER DEFAULT

Specifies the operating system user name that the database system uses for privilege checking.

You can specify a character string literal for the user name or you can specify the DEFAULT keyword. The DEFAULT keyword allows you to avoid placing the user name in a program's source code. If you specify the DEFAULT keyword, you pass the user name to the program by using a command line qualifier when you compile an SQL module or precompiled program. You use the USERNAME qualifier.

USING 'password'

USING DEFAULT

Specifies the user's password for the user name specified in the USER clause.

You can specify a character string literal for the PASSWORD or you can specify the DEFAULT keyword. The DEFAULT keyword allows you to avoid placing the user name in a program's source code. If you specify the DEFAULT keyword, you pass the password to the program by using a command line qualifier when you compile an SQL module or precompiled program. You use the PASSWORD qualifier.

DECLARE ALIAS Statement

Usage Notes

- DECLARE ALIAS is a nonexecutable statement that declares the database to the program at compilation. SQL does not attach to the database until it executes the first executable SQL statement in the program or SQL module.
- When SQL executes the first procedure in a module, by default it attaches to each alias in the module that is active.
- In interactive or dynamic SQL, you must use the ATTACH statement to add a database to the implicit environment. For more information, see the ATTACH Statement.
- The DECLARE ALIAS statements embedded in programs or in the DECLARE section of an SQL module must come before any DECLARE TRANSACTION or executable SQL statements. The DECLARE ALIAS statements tell the application what databases it can compile against.
- To use an alias with a multischema database, you must enable ANSI/ISO quoting and create a delimited identifier, as described in Section 2.2.11.
- You must ensure that the character sets specified by the DEFAULT CHARACTER SET and NATIONAL CHARACTER SET clauses are the same as the actual character sets of the database that is accessed at run time. If these character sets do not match, unexpected results occur at run time.
- The default character set specifies the character set for columns with CHAR and VARCHAR data types. For more information on the default character set, see Section 2.1.3.
- A national character set specifies the character set for columns with the NCHAR and NCHAR VARYING data types. For more information on the national character set, see Section 2.1.7.
- If the default character set is not specified in the DECLARE ALIAS statement, the default character set of the database file invoked at compile time is assumed.
- If the national character set is not specified in the DECLARE ALIAS statement, the national character set of the database file invoked at compile time is assumed.

DECLARE ALIAS Statement

- If the database default character set is not DEC_MCS, the PATHNAME specifier cannot be used due to a current limitation of the repository where object names must only contain DEC_MCS characters. SQL flags this as an error.

Examples

Example 1: Specifying a database and an alias in embedded SQL

This statement declares the database defined by the file specification personnel. The precompiler uses this definition when compiling the program and SQL uses the file personnel when the program runs. This name may be a logical name or the name portion of the file personnel.rdb.

```
EXEC SQL
    DECLARE PERS_ALIAS ALIAS FOR FILENAME personnel
END-EXEC
```

Example 2: Specifying a database with restricted access

This statement is the same as Example 1, but specifies restricted access to the database.

```
EXEC SQL
    DECLARE PERS_ALIAS ALIAS FOR FILENAME personnel
    RESTRICTED ACCESS
END-EXEC
```

Example 3: Specifying the DECLARE ALIAS statement

This portion of an application program declares the databases MIA1 and MIA_CHAR_SET. The precompiler uses the MIA1 database when compiling the program and SQL uses the MIA_CHAR_SET database when the program runs.

```
EXEC SQL
    DECLARE ALIAS
    COMPILETIME FILENAME MIA1
    RUNTIME FILENAME MIA_CHAR_SET
    DEFAULT CHARACTER SET DEC_KANJI
    NATIONAL CHARACTER SET KANJI;
```

Example 4: Specifying the DEFAULT user authentication

The following example shows how to use the DEFAULT clause for user name and password in an SQL module:

DECLARE ALIAS Statement

```
MODULE          TEST_DECLARE
DIAGNOSTICS     SQL99
LANGUAGE       C
PARAMETER      COLONS
ALIAS          RDB$DBHANDLE
-----
-----declarations-----
      DECLARE ALIAS COMPILETIME FILENAME mf_personnel
              USER DEFAULT
              USING DEFAULT
      RUNTIME :run_time_spec
      .
      .
      .
```

You pass the compile-time user name and password to the program by using command line qualifiers. For example, to compile the program use the following command line:

```
$ SQLMOD TESTDEC /USER=heleng /PASS= helenspasswd
```

At run time, the host language program can prompt the run-time user to specify only the file specification or the file specification and the user name and password at run time. The host language program can build the run time string.

For example, if the host language program uses only the file specification, the value of the variable passed to the program can be the following:

```
FILENAME "mf_personnel"
```

If the host language program uses the file specification, user name and password, the value of the variable passed to the program can be the following:

```
FILENAME "mf_personnel 'USER heleng' USING 'mypassword' "
```

You must enclose the string in quotation marks; whether you use single (') or double quotation marks (") depends upon the programming language.

If you use the following DECLARE ALIAS statement, the host language program can only prompt the run-time user to specify the file name.

```
      DECLARE ALIAS COMPILETIME FILENAME mf_personnel
              USER DEFAULT
              USING DEFAULT
      RUNTIME FILENAME :foo
```

DECLARE CURSOR Statement

Declares a cursor.

With cursors, the conditions that define the result table are specified by the select expression in the DECLARE CURSOR statement. SQL creates the result table when it executes an OPEN statement. The result table for a cursor exists until a CLOSE, COMMIT, or ROLLBACK statement executes, the program stops, or you exit from interactive SQL. However, the result table can exist across transactions if you define a **holdable cursor**. A holdable cursor can remain open and retain its position when a new SQL transaction begins.

Host language programs require cursors because programs must perform operations one row or element at a time, and therefore may execute statements more than once to process an entire result table or list.

The **scope** of a cursor describes the portion of a module or program where the cursor is valid. The **extent** of a cursor tells how long it is valid. All cursors in SQL have the scope of the entire module.

You can create three classes of cursors, depending on which DECLARE CURSOR statement you use:

- The DECLARE CURSOR statement is executed immediately. A cursor that you create with this statement, sometimes called a **static** cursor, exists only within the scope and extent of its module. Both the cursor name and SELECT statement are known to your application at compile time.
- The dynamic DECLARE CURSOR statement is executed immediately. The cursor name is known at compile time, and the SELECT statement is determined at run time. You must supply a name for the SELECT statement that is generated at run time. A dynamic cursor exists within the scope of its module, but its extent is the entire run of the program or image. For information about the dynamic DECLARE CURSOR statement, see the DECLARE CURSOR Statement, Dynamic.
- The extended dynamic DECLARE CURSOR statement must be precompiled or used as part of a procedure in an SQL module. You must supply parameters for the cursor name and for the identifier of a prepared SELECT statement that is generated at run time. An extended dynamic cursor exists within the scope and extent of the entire module. For information about the extended dynamic DECLARE CURSOR statement, see the DECLARE CURSOR Statement, Extended Dynamic .

DECLARE CURSOR Statement

Within each class, you can create two types of cursors:

- **Table cursors** are a method that SQL provides to access individual rows of a result table. (A **result table** is a temporary collection of columns and rows from one or more tables or views.)
- **List cursors** are a method that SQL provides to access individual elements in a list.

A **list** is an ordered collection of elements, or segments, of the data type LIST OF BYTE VARYING. For more information about the LIST OF BYTE VARYING data type, see Section 2.3.7.

List cursors enable users to scan through a very large data structure from within a language that does not provide support for objects of such size. Because lists exist as a set of elements within a row of a table, a list cursor must refer to a table cursor because the table cursor provides the row context.

Cursors are further divided according to the modes of operations that they can perform. Table cursors have four modes:

- **Update** cursors are the default table cursor. Rows are first read and locked for SHARED READ or PROTECTED READ and then later, when an UPDATE is performed, the rows are locked for EXCLUSIVE access. If the table is reserved for EXCLUSIVE access, the subsequent update lock is not required.
- **Read-only** cursors can be used to access row information from a result table whenever you do not intend to update the database. For example, you could use a read-only cursor to fetch row and column information for display.
- **Insert-only** cursors position themselves on a row that has just been inserted so that you can load lists into that row.
- **Update-only** cursors are used whenever you intend to modify many rows in the result table. When the UPDATE ONLY option is used, SQL uses a more aggressive lock mode that locks the rows for EXCLUSIVE access when first read. This mode avoids a lock promotion from SHARED READ or PROTECTED READ to EXCLUSIVE access. It may, therefore, avoid deadlocks normally encountered during the lock promotion.

DECLARE CURSOR Statement

List cursors have two modes:

- **Read-only** cursors are the default list cursor. They enable you to read existing lists. By adding the `SCROLL` keyword to the read-only list cursor clause, you enable Oracle Rdb to scroll forward and backward through the list segments as needed.
- **Insert-only** cursors enable you to insert data into a list.

Table 7–3 lists the classes, types, and modes of cursors that SQL provides.

Table 7–3 Classes, Types, and Modes of Cursors

DECLARE CURSOR		Dynamic DECLARE CURSOR		Extended Dynamic DECLARE CURSOR	
Table	List	Table	List	Table	List
Insert-only	Insert-only	Insert-only	Insert-only	Insert-only	Insert-only
Read-only	Read-only	Read-only	Read-only	Read-only	Read-only
Update-only		Update-only		Update-only	

For example, you must declare an insert-only table cursor to insert data into a table. If the table includes lists, use the table cursor to position on the correct row, and declare an insert-only list cursor to load the lists into that row. For details about using cursors to load data into your database, see the `INSERT Statement`.

To process the rows of a result table formed by a `DECLARE CURSOR` statement, you must use the `OPEN` statement to position the cursor before the first row. Subsequent `FETCH` statements retrieve the values of each row for display on the terminal or processing in a program. (You must close the cursor before you attempt to reopen it.) You can similarly process the elements of a list by using an `OPEN` statement to position the cursor before the first element in the list and repeating `FETCH` statements to retrieve successive elements.

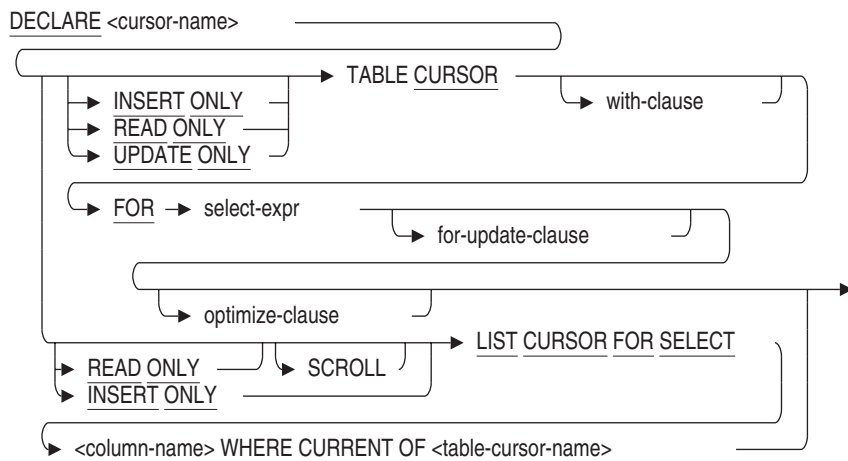
DECLARE CURSOR Statement

Environment

You can use the DECLARE CURSOR statement:

- In interactive SQL
- Embedded in host language programs to be precompiled
- As part of the DECLARE section in an SQL module
- In a context file

Format

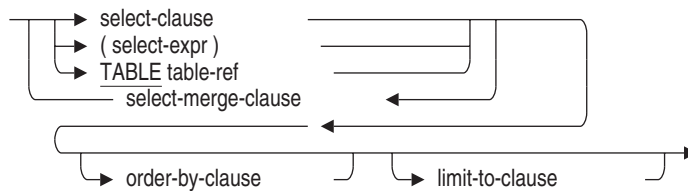


with-clause =

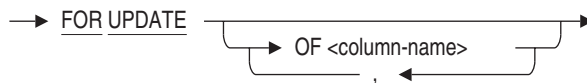


DECLARE CURSOR Statement

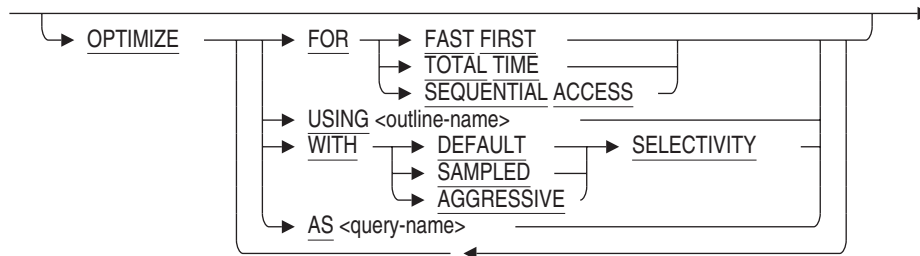
select-expr =



for-update-clause =



optimize-clause =



Arguments

cursor-name

Specifies the name of the cursor you want to declare. Use a name that is unique among all the cursor names in the module. Use any valid SQL name. See Section 2.2 for more information on user-supplied names.

You can use a parameter to specify the cursor name at run time in an extended dynamic DECLARE CURSOR statement. See the DECLARE CURSOR Statement, Extended Dynamic for more information on the extended dynamic DECLARE CURSOR statement.

DECLARE CURSOR Statement

FOR select-expr

A select expression that defines which columns and rows of which tables SQL includes in the cursor. See Section 2.8.1 for more information on select expressions.

FOR UPDATE OF column-name

Specifies the columns in a cursor that you or your program might later modify with an UPDATE statement. The column names in the FOR UPDATE clause must belong to a table or view named in the FROM clause.

You do not have to specify the FOR UPDATE clause of the DECLARE CURSOR statement to later modify rows using the UPDATE statement:

- If you do specify a FOR UPDATE clause and later specify columns in the UPDATE statement that are not in the FOR UPDATE clause, SQL issues a warning message and proceeds with the update modifications.
- If you do not specify a FOR UPDATE clause, you can update any column using the UPDATE statement. SQL does not issue any messages.

The FOR UPDATE OF clause in a SELECT statement provides UPDATE ONLY CURSOR semantics by locking all the rows selected.

INSERT ONLY

Specifies that a new list or a new row is created or opened.

If you specify a list cursor but do not specify the INSERT ONLY clause, SQL declares a read-only list cursor by default.

If you specify a table cursor but do not specify the INSERT ONLY clause, SQL declares an update cursor by default.

When you specify an insert-only cursor, all the value expressions in the select list must be read/write. When you declare an insert-only table cursor to insert lists, you must specify both table column and list column names in the FROM clause.

For more information about how to use insert-only cursors, see the INSERT Statement.

LIST CURSOR

Specifies a cursor that is used to manipulate columns of the data type LIST OF BYTE VARYING.

OPTIMIZE AS query-name

Assigns a name to the query. You must define the SET FLAGS 'STRATEGY' statement to see the access methods used to produce the results of the query.

DECLARE CURSOR Statement

OPTIMIZE FOR

The OPTIMIZE FOR clause specifies the preferred optimizer strategy for statements that specify a select expression. The following options are available:

- **FAST FIRST**

A query optimized for FAST FIRST returns data to the user as quickly as possible, even at the expense of total throughput.

If a query can be cancelled prematurely, you should specify FAST FIRST optimization. A good candidate for FAST FIRST optimization is an interactive application that displays groups of records to the user, where the user has the option of aborting the query after the first few screens. For example, singleton SELECT statements default to FAST FIRST optimization.

If optimization strategy is not explicitly set, FAST FIRST is the default.

- **TOTAL TIME**

If your application runs in batch, accesses all the records in the query, and performs updates or writes a report, you should specify TOTAL TIME optimization. Most queries benefit from TOTAL TIME optimization.

The following examples illustrate the DECLARE CURSOR syntax for setting a preferred optimization mode:

```
SQL> DECLARE TEMP1 TABLE CURSOR
cont> FOR
cont>   SELECT *
cont>     FROM EMPLOYEES
cont>     WHERE EMPLOYEE_ID > '00400'
cont> OPTIMIZE FOR FAST FIRST;
SQL> --
SQL> DECLARE TEMP2 TABLE CURSOR
cont> FOR
cont>   SELECT LAST_NAME, FIRST_NAME
cont>     FROM EMPLOYEES
cont>     ORDER BY LAST_NAME
cont> OPTIMIZE FOR TOTAL TIME;
```

- **SEQUENTIAL ACCESS**

Forces the use of sequential access. This is particularly valuable for tables that use the strict partitioning functionality.

OPTIMIZE USING outline-name

Explicitly names the query outline to be used with the select expression even if the outline IDs for the select expression and for the outline are different.

See the CREATE OUTLINE Statement for more information on creating an outline.

DECLARE CURSOR Statement

OPTIMIZE WITH

Selects one of three optimization controls: **DEFAULT** (as used by previous versions of Oracle Rdb), **AGGRESSIVE** (assumes smaller numbers of rows will be selected), and **SAMPLED** (which uses literals in the query to perform preliminary estimation on indices).

PRESERVE ON COMMIT

PRESERVE ON ROLLBACK

PRESERVE ALL

PRESERVE NONE

Specifies when a cursor remains open.

- **PRESERVE ON COMMIT**
On commit, all cursors close except those defined with the **WITH HOLD PRESERVE ON COMMIT** syntax. On rollback, all cursors close including those defined with the **WITH HOLD PRESERVE ON COMMIT** syntax.
This is the same as specifying the **WITH HOLD** clause without any preserve options.
- **PRESERVE ON ROLLBACK**
On rollback, all cursors close except those defined with the **WITH HOLD PRESERVE ON ROLLBACK** syntax. On commit, all cursors close including those defined with the **WITH HOLD PRESERVE ON ROLLBACK** syntax.
- **PRESERVE ALL**
All cursors remain open after commit or rollback. Cursors close with the **CLOSE** statement or when the session ends.
- **PRESERVE NONE**
All cursors close after a **CLOSE**, **COMMIT**, or **ROLLBACK** statement, when the program stops, or when you exit from interactive SQL.
This is the same as not specifying the **WITH HOLD** clause at all.

READ ONLY

Specifies that the cursor is not used to update the database.

SCROLL

Specifies that Oracle Rdb can read the items in a list from either direction (up or down) or at random. The **SCROLL** keyword must be used if the following fetch options are desired:

- **NEXT**

DECLARE CURSOR Statement

- PRIOR
- FIRST
- LAST
- RELATIVE
- ABSOLUTE

If SCROLL is not specified, the default for FETCH is NEXT. SCROLL is only supported for LIST cursors.

TABLE CURSOR

Specifies that the cursor you want to declare is a table cursor, rather than a list cursor. If you do not specify a cursor type, SQL declares a table cursor by default.

UPDATE ONLY

Specifies that the cursor is used to update the database.

Use an update-only cursor when you plan to update most of the rows you are fetching. The update-only cursor causes Oracle Rdb to apply more restrictive locking during the initial read operation, so that locks do not need to be upgraded later from READ to exclusive WRITE. This reduces the total number of lock requests per query, and may help to avoid deadlocks.

Use update-only table cursors to modify table rows. SQL does not allow update-only list cursors.

WHERE CURRENT OF table-cursor-name

Specifies the table cursor that provides the row context for the list cursor. The table cursor named must be defined using a DECLARE CURSOR statement.

WITH HOLD

Indicates that the cursor remain open and maintain its position after the transaction ends. This is called a **holdable cursor**.

Usage Notes

- You refer to cursors in INSERT, OPEN, CLOSE, FETCH, UPDATE, and DELETE statements. The order of those statements in a host language source file is not important; a CLOSE statement for a cursor can precede its corresponding OPEN statement so long as program control branches to process the OPEN statement first at run time. However, you must close a cursor before you reopen it.

DECLARE CURSOR Statement

- You can use the SQL CLOSE statement to close cursors individually, or use the sql_close_cursors() routine to close all open cursors. The sql_close_cursors() routine takes no arguments. For an example of this routine, see the *Oracle Rdb7 Guide to Distributed Transactions*.
- SQL does not restrict how many cursors you can have open at once. It is valid to declare and open more than one cursor at a time. However, if you plan to use static, dynamic, and extended dynamic cursors within the same program, you should avoid giving the same name to different cursors that share the same scope or extent.
- You cannot refer to list cursors in UPDATE or DELETE statements.
- SQL considers as read-only cursors those that:
 - Use the DISTINCT argument to eliminate duplicate rows from the result table
 - Name more than one table or view in the FROM clause
 - Include an aggregate function in the select list
 - Include a UNION, EXCEPT, or INTERSECT operator in the main query
 - Contain a GROUP BY or HAVING clause in the main query

When a cursor is declared as READ ONLY, it can never be referenced in a positional UPDATE or DELETE statement or an INSERT INTO cursor-name statement.

When a cursor has neither INSERT ONLY, READ ONLY, or UPDATE ONLY specified, it is considered a general cursor that can be used for a DELETE, INSERT or UPDATE statement. However, if any of the above listed items occurs, SQL implicitly considers the cursor to be a READ ONLY cursor.

- You can process a table cursor only in the forward direction. If you want to move the table cursor back to a row that you already processed, you must close the table cursor and open it again.
- The order of the result table is unpredictable unless you specify an ORDER BY clause in the DECLARE CURSOR statement. (The ORDER BY clause is not valid in a list cursor declaration.)
- SQL evaluates the result table of the cursor (specified by the SELECT statement) when it executes an OPEN statement for the cursor.

DECLARE CURSOR Statement

- SQL evaluates any parameters in the select expression of a DECLARE CURSOR statement when it executes the OPEN statement for the cursor. It cannot evaluate the parameters again until you close and open the cursor again.
- If a DECLARE CURSOR statement contains parameters, you pass the parameters to it by declaring them in the procedure that contains the OPEN statement. In addition, you must specify the parameter in the host language call to the procedure that contains the OPEN statement. Because the DECLARE CURSOR statement appears in the declaration section of a module, not a procedure, you cannot pass the parameters directly to the DECLARE CURSOR statement.

For examples of declaring cursors with parameters and passing parameters to an SQL module, see Chapter 3.

- You cannot refer to insert-only cursors in the following statements:
 - DELETE and UPDATE statements that specify the CURRENT OF clause
 - FETCH statements
- You cannot use the INSERT ONLY clause in a DECLARE CURSOR statement that contains one or more of the following clauses in the main query:
 - DISTINCT
 - WHERE
 - ORDER BY
 - GROUP BY
 - UNION, EXCEPT (MINUS), INTERSECT
- You can use only an insert-only cursor for the cursor name in an INSERT statement used to add a new row to a table cursor or a new element to a list cursor.
- When you define an insert-only table cursor, you must include the LIST column in the select list of the table cursor. For an example, see Example 3.
- A DECLARE CURSOR statement that uses parameters to specify statements and cursor names is an extended dynamic DECLARE CURSOR statement. An extended dynamic DECLARE CURSOR statement lets programs supply cursor and statement names at run time. See the

DECLARE CURSOR Statement

DECLARE CURSOR Statement, Extended Dynamic for more information on the extended dynamic DECLARE CURSOR statement.

An extended dynamic DECLARE CURSOR statement is an executable statement and returns a status value. In the module language, you must include such a statement in a procedure.

- When accessing list data, you must be careful to close a list cursor before you fetch the next row in the table cursor. If you fetch some, but not all, rows from a list cursor and move to the next row in the table cursor without closing the list cursor, you continue to fetch rows from the previous list cursor. SQL does not issue a warning or error message telling you that you opened two list cursors.

```
SQL> -- Define a cursor of Board Manufacturing Department Managers:
SQL> --
SQL> DECLARE BM_MGR CURSOR FOR
cont> SELECT EMPLOYEE_ID, RESUME FROM RESUMES R, CURRENT_INFO CI
cont> WHERE R.EMPLOYEE_ID = CI.ID AND DEPARTMENT
cont> CONTAINING "BOARD MANUFACTURING" AND JOB = "Department Manager";
SQL> --
SQL> -- Define a cursor for resumes of those managers:
SQL> DECLARE THE_RESUME LIST CURSOR FOR
cont> SELECT RESUME WHERE CURRENT OF BM_MGR;
SQL> --
SQL> -- Build the manager's cursor:
SQL> OPEN BM_MGR;
SQL> --
SQL> -- Fetch the manager's row:
SQL> FETCH BM_MGR;
  R.EMPLOYEE_ID  R.RESUME
  00164                72:2:3
SQL> --
SQL> -- Get part of the resume:
SQL> OPEN THE_RESUME;
SQL> FETCH THE_RESUME;
  RESUME
  This is the resume for Alvin Toliver
SQL> --
SQL> -- Do not close the resume, and access the next manager:
SQL> FETCH BM_MGR;
  R.EMPLOYEE_ID  R.RESUME
  00166                72:2:9
SQL> -- SQL continues to fetch from Toliver's resume (00164)
SQL> -- because the list cursor was not closed.
SQL> -- If it were a new resume, you would see
SQL> -- a new "This is the resume for ..." line.
SQL> FETCH THE_RESUME;
  RESUME
  Boston, MA
```

DECLARE CURSOR Statement

- The declared cursor must refer to the same table or list of tables specified in a SET TRANSACTION RESERVING clause or the LOCK TABLE statement. For example:

```
SQL> SET TRANSACTION RESERVING jobs FOR WRITE;
SQL> DECLARE curs1 CURSOR WITH HOLD FOR
cont> SELECT first_name,last_name FROM employees;
SQL> OPEN CURS1;
%RDB-E-UNRES_REL, relation EMPLOYEES in specified request is not a relation
reserved in specified transaction
```

- You can specify only the WITH HOLD clause for table cursors.
- It is possible in some queries for Rdb to prefetch the data for the cursor during the OPEN statement. Examples include cursors that include the ORDER BY clause which will require the data to be read and sorted before delivering the first row of the result. The rows that are fetched are now a snapshot of the data at the time of the OPEN and may become obsolete after the COMMIT statement has executed.

For example, user BROWN declares and opens a cursor accessing the employees table and later commits the transaction, but the WITH HOLD cursor remains open. User JONES deletes an employee from the employees table during the time BROWN has the cursor open. BROWN still sees the employee deleted by JONES because BROWN is accessing a temporary copy from the original state of table.

- You can define an SQL session default setting for holdable cursors using the SET HOLD CURSORS statement. See the SET HOLD CURSORS Statement for more information.
- The WITH HOLD PRESERVE ALL clause conforms to the ODBC driver behavior of cursors.
- If an outline exists, Oracle Rdb uses the outline specified in the OPTIMIZE USING clause unless one or more of the directives in the outline cannot be followed. For example, if the compliance level for the outline is mandatory and one of the indexes specified in the outline directives has been deleted, the outline is not used. SQL issues an error message if an existing outline cannot be used.

If you specify the name of an outline that does not exist, Oracle Rdb compiles the query, ignores the outline name, and searches for an existing outline with the same outline ID as the query. If an outline with the same outline ID is found, Oracle Rdb attempts to execute the query using the directives in that outline. If an outline with the same outline ID is not found, the optimizer selects a strategy for the query for execution.

DECLARE CURSOR Statement

See the *Oracle Rdb7 Guide to Database Performance and Tuning* for more information regarding query outlines.

Examples

Example 1: Declaring a table cursor in interactive SQL

The following example declares a cursor named SALARY_INFO. The result table for SALARY_INFO contains the names and current salaries of employees and is sorted by last name.

```
SQL> --
SQL> DECLARE SALARY_INFO CURSOR FOR
cont> SELECT  E.FIRST_NAME, E.LAST_NAME, S.SALARY_AMOUNT
cont> FROM    EMPLOYEES E, SALARY_HISTORY S
cont> WHERE   E.EMPLOYEE_ID = S.EMPLOYEE_ID
cont>        AND
cont>        S.SALARY_END IS NULL
cont> ORDER BY
cont>        E.LAST_NAME ASC;
SQL> --
SQL> -- Use an OPEN statement to open the cursor and
SQL> -- position it before the first row of the
SQL> -- result table:
SQL> OPEN SALARY_INFO;
SQL> --
SQL> -- Finally, use two FETCH statements to see the
SQL> -- first two rows of the cursor:
SQL> FETCH SALARY_INFO;
  E.FIRST_NAME  E.LAST_NAME      S.SALARY_AMOUNT
  Louie         Ames              $26,743.00
SQL> FETCH SALARY_INFO;
  E.FIRST_NAME  E.LAST_NAME      S.SALARY_AMOUNT
  Leslie        Andriola          $50,424.00
```

Example 2: Declaring a table cursor in a C program

This simple program uses embedded DECLARE CURSOR, OPEN, and FETCH statements to retrieve and print the names and departments of managers.

```
#include <stdio.h>

void main ()
{
  int SQLCODE;
  char FNAME[15];
  char LNAME[15];
  char DNAME[31];
```

DECLARE CURSOR Statement

```
/* Declare the cursor: */
exec sql
    DECLARE MANAGER CURSOR FOR
        SELECT E.FIRST_NAME, E.LAST_NAME, D.DEPARTMENT_NAME
        FROM EMPLOYEES E, DEPARTMENTS D
        WHERE E.EMPLOYEE_ID = D.MANAGER_ID ;

/* Open the cursor: */
exec sql
    OPEN MANAGER;

/* Start a loop to process the rows of the cursor: */
for (;;)
{
    /* Retrieve the rows of the cursor
    and put the value in host language variables: */
    exec sql
        FETCH MANAGER INTO :FNAME, :LNAME, :DNAME;
    if (SQLCODE != 0) break;
    /* Print the values in the variables: */
    printf ("%s %s %s\n", FNAME, LNAME, DNAME);
}

/* Close the cursor: */
exec sql
    CLOSE MANAGER;
}
```

Example 3: Using table and list cursors to retrieve list data in interactive SQL

The following example declares a table and list cursor to retrieve list information:

```
SQL> DECLARE TBLCURSOR INSERT ONLY TABLE CURSOR FOR
cont> SELECT EMPLOYEE_ID, RESUME FROM RESUMES;
SQL> DECLARE LSTCURSOR INSERT ONLY LIST CURSOR FOR SELECT RESUME WHERE
CURRENT OF TBLCURSOR;
SQL> OPEN TBLCURSOR;
SQL> INSERT INTO CURSOR TBLCURSOR (EMPLOYEE_ID) VALUES ('00164');
1 row inserted
SQL> OPEN LSTCURSOR;
SQL> INSERT INTO CURSOR LSTCURSOR VALUES ('This is the resume for 00164');
SQL> INSERT INTO CURSOR LSTCURSOR VALUES ('Boston, MA');
SQL> INSERT INTO CURSOR LSTCURSOR VALUES ('Oracle Corporation');
SQL> CLOSE LSTCURSOR;
SQL> CLOSE TBLCURSOR;
SQL> COMMIT;
SQL> DECLARE TBLCURSOR2 CURSOR FOR SELECT EMPLOYEE_ID,
cont> RESUME FROM RESUMES;
SQL> DECLARE LSTCURSOR2 LIST CURSOR FOR SELECT RESUME WHERE
CURRENT OF TBLCURSOR2;
```

DECLARE CURSOR Statement

```
SQL> OPEN TBLCURSOR2;
SQL> FETCH TBLCURSOR2;
    00164
SQL> OPEN LSTCURSOR2;
SQL> FETCH LSTCURSOR2;
RESUME
This is the resume for 00164
SQL> FETCH LSTCURSOR2;
RESUME
Boston, MA
SQL> FETCH LSTCURSOR2;
RESUME
Oracle Corporation
SQL> FETCH LSTCURSOR2;
RESUME
%RDB-E-STREAM_EOF, attempt to fetch past end of record stream
SQL> CLOSE LSTCURSOR2;
SQL> SELECT * FROM RESUMES;
  EMPLOYEE_ID  RESUME
  00164                1:701:2
1 row selected
SQL> CLOSE TBLCURSOR2;
SQL> COMMIT;
```

Example 4: Using the scroll attribute for a list cursor

The following example declares a table and read-only scrollable list cursor to retrieve list information by scrolling back and forth between segments of the list:

```
SQL> DECLARE CURSOR_ONE
cont>   TABLE CURSOR FOR
cont>   (SELECT EMPLOYEE_ID,RESUME FROM RESUMES);
SQL> --
SQL> DECLARE CURSOR_TWO
cont>   READ ONLY
cont>   SCROLL
cont>   LIST CURSOR
cont>   FOR SELECT RESUME
cont>   WHERE CURRENT OF CURSOR_ONE;
```

DECLARE CURSOR Statement

Example 5: Declaring a holdable cursor

```
SQL> -- Declare a holdable cursor that remains open on COMMIT
SQL> --
SQL> DECLARE curs1 CURSOR
cont>     WITH HOLD PRESERVE ON COMMIT
cont>     FOR SELECT e.first_name, e.last_name
cont>     FROM employees e
cont>     ORDER BY e.last_name;
SQL> OPEN curs1;
SQL> FETCH curs1;
FIRST_NAME  LAST_NAME
Louie       Ames
SQL> FETCH curs1;
FIRST_NAME  LAST_NAME
Leslie      Andriola
SQL> COMMIT;
SQL> FETCH curs1;
FIRST_NAME  LAST_NAME
Joseph      Babbins
SQL> FETCH curs1;
FIRST_NAME  LAST_NAME
Dean        Bartlett
SQL> ROLLBACK;
SQL> FETCH curs1;
%SQL-F-CURNOTOPE, Cursor CURS1 is not opened
SQL> --
SQL> -- Declare another holdable cursor that remains open always
SQL> --
SQL> DECLARE curs2 CURSOR
cont>     WITH HOLD PRESERVE ALL
cont>     FOR SELECT e.first_name, e.last_name
cont>     FROM employees e
cont>     ORDER BY e.last_name;
SQL> OPEN curs2;
SQL> FETCH curs2;
FIRST_NAME  LAST_NAME
Louie       Ames
SQL> FETCH curs2;
FIRST_NAME  LAST_NAME
Leslie      Andriola
SQL> COMMIT;
SQL> FETCH curs2;
FIRST_NAME  LAST_NAME
Joseph      Babbins
SQL> FETCH curs2;
FIRST_NAME  LAST_NAME
Dean        Bartlett
SQL> ROLLBACK;
SQL> FETCH curs2;
FIRST_NAME  LAST_NAME
Wes         Bartlett
```

DECLARE CURSOR Statement, Dynamic

DECLARE CURSOR Statement, Dynamic

Declares a cursor where the `SELECT` statement is supplied at run time in a parameter.

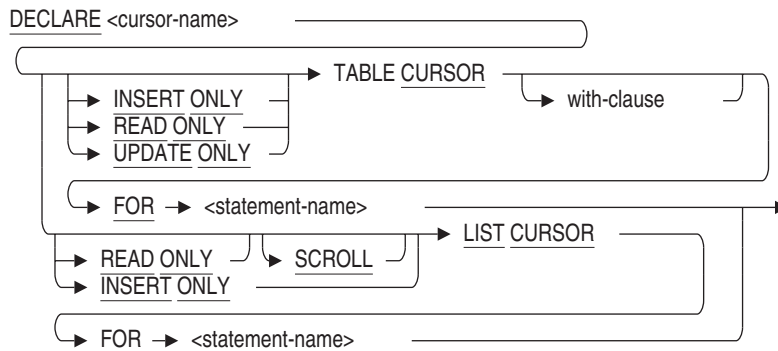
Refer to the `DECLARE CURSOR` Statement for a detailed description of statement elements that apply to both dynamic and nondynamic `DECLARE CURSOR` statements.

Environment

You can use the dynamic `DECLARE CURSOR` statement:

- Embedded in host language programs to be precompiled
- As part of the `DECLARE` statement section in an SQL module

Format



with-clause =



DECLARE CURSOR Statement, Dynamic

Arguments

cursor-name

The name of the cursor you want to declare. Use a name that is unique among all the cursor names in the module. Use any valid SQL name. See Section 2.2 for more information on identifiers.

FOR statement-name

A name that identifies a prepared SELECT statement that is generated at run time.

INSERT ONLY

Specifies that a new list or a new row is created or opened.

LIST CURSOR

Specifies that you are declaring a cursor to access the elements in a list.

PRESERVE ON COMMIT**PRESERVE ON ROLLBACK****PRESERVE ALL****PRESERVE NONE**

Specifies when a cursor remains open.

- **PRESERVE ON COMMIT**
On commit, all cursors close except those defined with the WITH HOLD PRESERVE ON COMMIT syntax. On rollback, all cursors close including those defined with the WITH HOLD PRESERVE ON COMMIT syntax.
This is the same as specifying the WITH HOLD clause without any preserve options.
- **PRESERVE ON ROLLBACK**
On rollback, all cursors close except those defined with the WITH HOLD PRESERVE ON ROLLBACK syntax. On commit, all cursors close including those defined with the WITH HOLD PRESERVE ON ROLLBACK syntax.
- **PRESERVE ALL**
All cursors remain open after commit or rollback. Cursors close with the CLOSE statement or when the session ends.
- **PRESERVE NONE**
All cursors close after a CLOSE, COMMIT, or ROLLBACK statement, when the program stops, or when you exit from interactive SQL.

DECLARE CURSOR Statement, Dynamic

This is the same as not specifying the WITH HOLD clause at all.

READ ONLY

Specifies that the cursor is not used to update the database.

SCROLL

Specifies that Oracle Rdb can read the items in a list from either direction (up or down) or at random.

TABLE CURSOR

Specifies that you are declaring a cursor to access the rows in a table.

UPDATE ONLY

Specifies that the cursor is used to update the database.

WITH HOLD

Indicates that the cursor remain open and maintain its position after the transaction ends. This is called a **holdable cursor**.

Usage Notes

- In a dynamic DECLARE CURSOR statement, the cursor name is compiled, but the SELECT statement is determined at run time.
- Because a dynamic DECLARE CURSOR statement is not executable, you must place this statement in the DECLARE section of an SQL module, as with static DECLARE CURSOR statements.
- Cursors and views that contain a GROUP BY, UNION, EXCEPT (MINUS), or INTERSECT clause in their main query cannot be accessed using dynamic cursors which require access by DBKEY. If a user attempts to access one of these views with a dynamic cursor, the following error is returned when the cursor is opened:

"RDMS-F-VIEWNORET, view cannot be retrieved by database key".

The workaround for this problem is to use nondynamic cursors to access the view. If a dynamic cursor must be used, the statement should access the base tables that make up the view (with the GROUP BY and UNION clauses, as appropriate) and not the view itself.

- Refer also to the Usage Notes for the DECLARE CURSOR statement.

DECLARE CURSOR Statement, Dynamic

Examples

Example 1: Using a parameter for a statement name

```

    .
    .
    .
* This program prepares a statement for dynamic execution from the string
* passed to it, and uses a dynamic cursor to fetch a row from a table.
*
*/
#include <stdio.h>
#include <descrip.h>

struct SQLDA_STRUCT {
    char SQLDAID[8];
    int SQLDABC;
    short SQLN;
    short SQLD;
    struct {
        short SQLTYPE;
        short SQLLEN;
        char *SQLDATA;
        short *SQLIND;
        short SQLNAME_LEN;
        char SQLNAME[30];
        } SQLVAR[];
    } *SQLDA;

main()
{
    /*
     * General purpose locals
     */
    int    i;
    long   sqlcode;

    char   command_string[256];

    /*
     * Allocate SQLDA structures.
     */

    SQLDA = malloc(500);
    SQLDA->SQLN = 20;

    /* Get the SELECT statement at run time. */

    printf("\n Enter a SELECT statement.\n");
    printf("\n Do not end the statement with a semicolon.\n");
    gets(command_string);
}
```

DECLARE CURSOR Statement, Dynamic

```
/* Prepare the SELECT statement. */
PREP_STMT( &sqlcode, &command_string, SQLDA );
if (sqlcode != 0)
    goto err;

/* Open the cursor. */
OPEN_CURSOR( &sqlcode );
if (sqlcode != 0)
    goto err;

/* Allocate memory. */
for (i=0; i < SQLDA->SQLD; i++) {
    SQLDA->SQLVAR[i].SQLDATA = malloc( SQLDA->SQLVAR[i].SQLLEN );
    SQLDA->SQLVAR[i].SQLIND = malloc( 2 );
}

/* Fetch a row. */
FETCH_CURSOR( &sqlcode, SQLDA );
if (sqlcode != 0)
    goto err;

/* Use the SQLDA to determine the data type of each column in the row
and print the column. For simplicity, test for only two data types.
CHAR and INT. */
for (i=0; i < SQLDA->SQLD; i++) {
    switch (SQLDA->SQLVAR[i].SQLTYPE) {
        case SQLDA_CHAR; /* Character */
            printf( "%s", SQLDA->SQLVAR[i].SQLDATA );
            break;
        case SQLDA_INTEGER: /* Integer */
            printf( "%d", SQLDA->SQLVAR[i].SQLDATA );
            break;
        default:
            printf( "Some other datatype encountered\n" );
    }
}

/* Close the cursor. */
CLOSE_CURSOR( &sqlcode );

ROLLBACK(&sqlcode );
return;
.
.
.
}
```

DECLARE CURSOR Statement, Dynamic

Example 2: SQL module file that the preceding program calls

```
-- This program uses dynamic cursors to fetch a row.
--
--
MODULE          C_MOD_DYN_CURS
LANGUAGE        C
AUTHORIZATION   RDB$DBHANDLE

DECLARE ALIAS FOR FILENAME personnel

-- Declare the dynamic cursor. Use a statement name to identify a
-- prepared SELECT statement.

DECLARE CURSOR1 CURSOR FOR STMT_NAME

-- Prepare the statement from a statement entered at run time
-- and specify that SQL write information about the number and
-- data type of select list items to the SQLDA.

PROCEDURE PREP_STMT
    SQLCODE
    COMMAND_STRING CHAR (256)
    SQLDA;

    PREPARE STMT_NAME SELECT LIST INTO SQLDA FROM COMMAND_STRING;

PROCEDURE OPEN_CURSOR
    SQLCODE;

    OPEN CURSOR1;

PROCEDURE FETCH_CURSOR
    SQLCODE
    SQLDA;

    FETCH CURSOR1 USING DESCRIPTOR SQLDA;

PROCEDURE CLOSE_CURSOR
    SQLCODE;

    CLOSE CURSOR1;

PROCEDURE ROLLBACK
    SQLCODE;

    ROLLBACK;
```

DECLARE CURSOR Statement, Extended Dynamic

DECLARE CURSOR Statement, Extended Dynamic

Declares an extended dynamic cursor. An extended dynamic DECLARE CURSOR statement is a DECLARE CURSOR statement in which both the cursor name and the SELECT statement are supplied in parameters at run time.

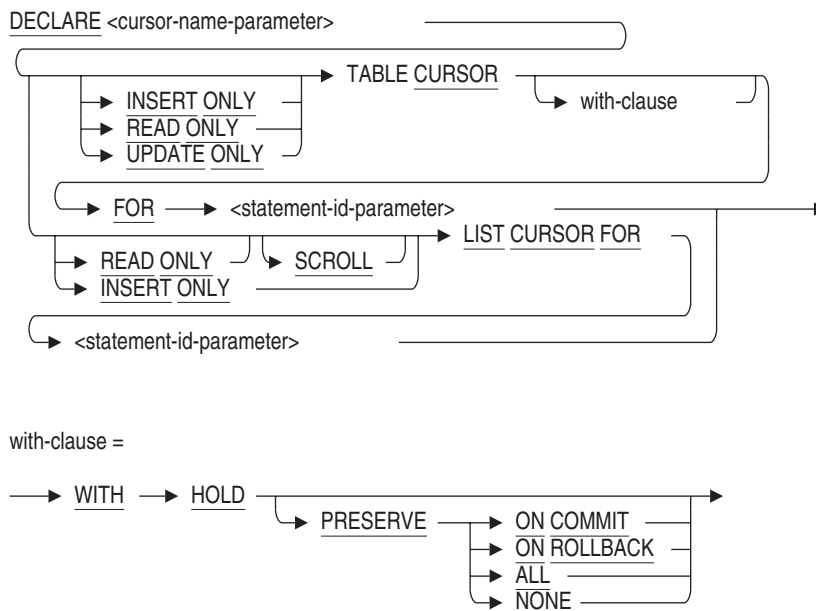
See the DECLARE CURSOR Statement for a detailed description of statement elements that apply to both dynamic and nondynamic DECLARE CURSOR statements.

Environment

You can use the extended dynamic DECLARE CURSOR statement:

- Embedded in host language programs to be precompiled
- As part of a procedure in an SQL module

Format



DECLARE CURSOR Statement, Extended Dynamic

Arguments

cursor-name-parameter

Contains the name of the cursor you want to declare. Use a character string parameter to hold the cursor name that the program supplies at run time.

FOR statement-id-parameter

A parameter that contains an integer that identifies a prepared SELECT statement. Use an integer parameter to hold the statement identifier that SQL generates and assigns to the parameter when SQL executes a PREPARE statement.

INSERT ONLY

Specifies that a new list or a new row is created or opened.

LIST CURSOR FOR

Specifies that you are declaring a cursor to access the elements in a list.

PRESERVE ON COMMIT

PRESERVE ON ROLLBACK

PRESERVE ALL

PRESERVE NONE

Specifies when a cursor remains open.

- **PRESERVE ON COMMIT**
On commit, all cursors close except those defined with the WITH HOLD PRESERVE ON COMMIT syntax. On rollback, all cursors close including those defined with the WITH HOLD PRESERVE ON COMMIT syntax.
This is the same as specifying the WITH HOLD clause without any preserve options.
- **PRESERVE ON ROLLBACK**
On rollback, all cursors close except those defined with the WITH HOLD PRESERVE ON ROLLBACK syntax. On commit, all cursors close including those defined with the WITH HOLD PRESERVE ON ROLLBACK syntax.
- **PRESERVE ALL**
All cursors remain open after commit or rollback. Cursors close with the CLOSE statement or when the session ends.
- **PRESERVE NONE**
All cursors close after a close, commit, or rollback statement, when the program stops, or when you exit from interactive SQL.

DECLARE CURSOR Statement, Extended Dynamic

This is the same as not specifying the WITH HOLD clause at all.

READ ONLY

Specifies that the cursor is not used to update the database.

SCROLL

Specifies that Oracle Rdb can read the items in a list from either direction (up or down) or at random.

TABLE CURSOR FOR

Specifies that you are declaring a cursor to access the rows in a table.

UPDATE ONLY

Specifies that the cursor is used to update the database.

WITH HOLD

Indicates that the cursor remain open and maintain its position after the transaction ends. This is called a **holdable cursor**.

Usage Notes

- An extended dynamic DECLARE CURSOR statement is an executable statement in dynamic SQL. It lets you specify, through parameters, both the name of a cursor and the identifier of the SELECT statement on which the cursor is based at run time. In general, using extended dynamic SQL allows a single set of SQL procedures to concurrently control an arbitrary number of prepared statements.
- The extended dynamic DECLARE CURSOR statement lets you use one DECLARE CURSOR-PREPARE statement combination for multiple, dynamically generated SELECT statements. This eliminates the necessity of coding a DECLARE CURSOR and PREPARE statement for each dynamically generated SELECT statement.
- You must use parameters to specify both the cursor name and the statement identifier in an extended dynamic DECLARE CURSOR statement. Specifying either the cursor name or the statement identifier explicitly but not both through a parameter generates an error. Specifying both the cursor name and statement identifier explicitly makes the cursor a nondynamic cursor and the DECLARE CURSOR statement a nonexecutable statement.

DECLARE CURSOR Statement, Extended Dynamic

- Because an extended dynamic DECLARE CURSOR statement is executable, it returns an execution status (SQLSTATE or SQLCODE) at run time. Your program should check the status after executing an extended dynamic DECLARE CURSOR statement.

Because an extended dynamic DECLARE CURSOR statement is executable, you must place this statement in programs and SQL module files where executable statements are allowed. For example, you must place extended dynamic DECLARE CURSOR statements within a procedure in an SQL module, not in the DECLARE section as with static or dynamic DECLARE CURSOR statements.

- Refer also to the Usage Notes for the DECLARE CURSOR statement.

Example

Example 1: Using parameters for statement and cursor names

The following example shows two procedures from the online sample program SQL\$MULTI_STMT_DYN.SQLADA. These procedures show the use of parameters for statement and cursor names.

```
.
.
.
-- This procedure prepares a statement for dynamic execution from the string
-- passed to it. This procedure can prepare any number of statements
-- because the statement is passed to it as the parameter, cur_procid.
procedure PREPARE_SQL is
  CUR_CURSOR : string(1..31) := (others => ' ');
  CUR_PROCID : integer := 0;
  CUR_STMT   : string(1..1024) := (others => ' ');
begin
  -- Allocate separate SQLDAs for parameter markers (sqlda_in) and select list
  -- items (sqlda_out). Assign the value of the constant MAXPARMS (set in the
  -- declarations section) to the SQLN field of both SQLDA structures. SQLN
  -- specifies to SQL the maximum size of the SQLDA.
  sqlda_in := new sqlda_record;
  sqlda_in.sqln := maxparms;
  sqlda_out := new sqlda_record;
  sqlda_out.sqln := maxparms;
  -- Assign the SQL statement that was constructed in the procedure
  -- CONSTRUCT_SQL to the variable cur_stmt.
  cur_stmt := sql_stmt;
```

DECLARE CURSOR Statement, Extended Dynamic

```
-- Use the PREPARE...SELECT LIST statement to prepare the dynamic statement
-- and write information about any select list items in it to sqlda_out.
-- It prepares a statement for dynamic execution from the string passed to
-- it. It also writes information about the number and data type of any
-- select list items in the statement to an SQLDA (specifically, the
-- sqlda_out SQLDA specified).
--
-- Note that the PREPARE statement could have prepared the statement without
-- writing to an SQLDA. Instead, a separate DESCRIBE...SELECT LIST statement
-- would have written information about any select list items to an SQLDA.
EXEC SQL PREPARE :cur_procid SELECT LIST INTO :sqlda_out FROM :cur_stmt;
case sqlca.sqlcode is
    when sql_success => null;
    when others => raise syntax_error;
end case;

-- Use the DESCRIBE...MARKERS statement to write information about any
-- parameter markers in the dynamic statement to sqlda_in. This statement
-- writes information to an SQLDA (specifically, the sqlda_in SQLDA
-- specified) about the number and data type of any parameter markers in
-- the prepared dynamic statement. Note that SELECT statements may also
-- have parameter markers.

EXEC SQL DESCRIBE :cur_procid MARKERS INTO sqlda_in;
case sqlca.sqlcode is
    when sql_success => null;
    when others => raise syntax_error;
end case;

-- If the operation is "Read," create a unique name for the cursor name
-- so that the program can pass the cursor name to the dynamic DECLARE
-- CURSOR statement.

if cur_op(1) = 'R' then
    cur_cursor(1) := 'C';
    cur_cursor(2..name_strlng) := cur_name(1..name_strlng - 1);

-- Declare the dynamic cursor.

    EXEC SQL DECLARE :cur_cursor CURSOR FOR :cur_procid;
    case sqlca.sqlcode is
        when sql_success => null;
        when others => raise syntax_error;
    end case;
end if;

number_of_procs := number_of_procs + 1;
sqlda_in_array(number_of_procs) := sqlda_in;
sqlda_out_array(number_of_procs) := sqlda_out;
procedure_names(number_of_procs) := cur_name;
procedure_ids(number_of_procs) := cur_procid;
if cur_op(1) = 'R' then
    cursor_names(number_of_procs) := cur_cursor;
end if;
```

DECLARE CURSOR Statement, Extended Dynamic

```
exception
  when syntax_error =>
    sql_get_error_text(get_error_buffer,get_error_length);
    put_line(get_error_buffer(1..integer(get_error_length)));
    put("Press RETURN to continue. ");
    get_line(terminal,release_screen,last);
    new_line;
end PREPARE_SQL;
.
.
.
begin -- procedure body DISPLAY_DATA
-- Before displaying any data, allocate buffers to hold the data
-- returned by SQL.
--
  allocate_buffers;

-- Allocate and assign SQLDAs for the requested SQL procedure.
--
sqlda_in := new sqlda_record;
sqlda_in := sqlda_in_array(stmt_index);
sqlda_out := new sqlda_record;
sqlda_out := sqlda_out_array(stmt_index);
cur_cursor := cursor_names(stmt_index);
-- Open the previously declared cursor. The statement specifies
-- an SQLDA (specifically, sqlda_in) as the source of addresses for any
-- parameter markers in the cursor's SELECT statement.
--
EXEC SQL OPEN :cur_cursor USING DESCRIPTOR sqlda_in;
case sqlca.sqlcode is
  when sql_success => null;
  when others => raise unexpected_error;
end case;

-- Fetch the first row from the result table. This statement fetches a
-- row from the opened cursor and writes it to the addresses specified
-- in an SQLDA (specifically, sqlda_out).
--
EXEC SQL FETCH :cur_cursor USING DESCRIPTOR sqlda_out;
case sqlca.sqlcode is
-- Check to see if the result table has any rows.
  when sql_success => null;
  when stream_eof =>
    put_line("No records found.");
    new_line;
  when others => raise unexpected_error;
end case;

-- Set up a loop to display the first row, then fetch and display second
-- and subsequent rows.
```

DECLARE CURSOR Statement, Extended Dynamic

```
    rowcount := 0;
    while sqlca.sqlcode = 0 loop
        rowcount := rowcount + 1;
--      Execute the DISPLAY_ROW procedure.
        display_row;
--      To only display 5 rows, exit the loop if the loop counter
--      equals MAXROW (coded as 5 in this program).
        if rowcount = maxrows then exit; end if;
--      Fetch another row, exit the loop if no more rows.
        EXEC SQL FETCH :cur_cursor USING DESCRIPTOR sqlda_out;
        case sqlca.sqlcode is
            when sql_success => null;
            when stream_eof => exit;
            when others => raise unexpected_error;
        end case;
    end loop;
-- Close the cursor.
EXEC SQL CLOSE :cur_cursor;
case sqlca.sqlcode is
    when sql_success => null;
    when others => raise unexpected_error;
end case;
exception
    when unexpected_error =>
        sql_get_error_text(get_error_buffer,get_error_length);
        EXEC SQL ROLLBACK;
        put_line("This condition was not expected.");
        put_line(get_error_buffer(1..integer(get_error_length)));
        put("Press RETURN to continue. ");
        get_line(terminal,release_screen,last);

-- Stop and let the user look before returning.
    skip;
    put_line("Press RETURN to proceed. ");
    get_line(terminal,release_screen,last);
end DISPLAY_DATA;
```

DECLARE FUNCTION Statement

DECLARE FUNCTION Statement

Declares an external function interface for use in database definition statements.

The DECLARE FUNCTION statement is documented under the DECLARE Routine Statement. For complete information on declaring a procedure, see the DECLARE Routine Statement.

DECLARE LOCAL TEMPORARY TABLE Statement

DECLARE LOCAL TEMPORARY TABLE Statement

Explicitly declares a local temporary table.

The metadata for a declared local temporary table is not stored in the database and cannot be shared by other modules. These tables are sometimes called **scratch tables**.

The data stored in the table cannot be shared between SQL sessions or modules in a single session. Unlike persistent base tables, the metadata and data do not persist beyond an SQL session.

In addition to declared local temporary tables, there are two other types of temporary tables:

- Global temporary tables
- Local temporary tables

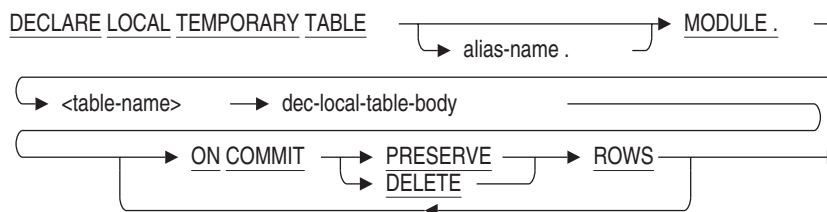
See the CREATE TABLE Statement for additional information on global and local temporary tables.

Environment

You can use the DECLARE LOCAL TEMPORARY TABLE statement:

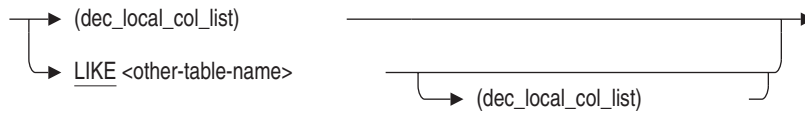
- In interactive SQL
- In dynamic SQL as a statement to be dynamically executed
- In a stored module

Format

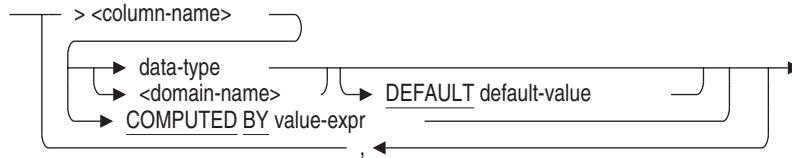


DECLARE LOCAL TEMPORARY TABLE Statement

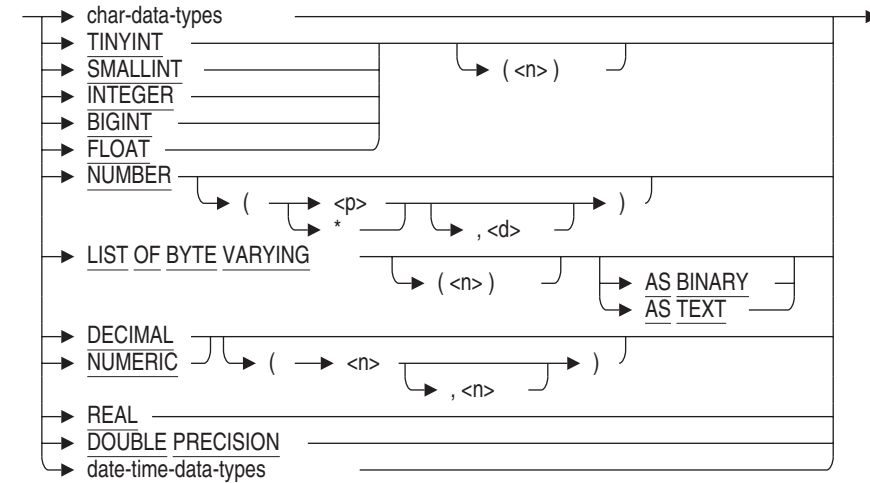
dec_local_table_body



dec-local-col-list =

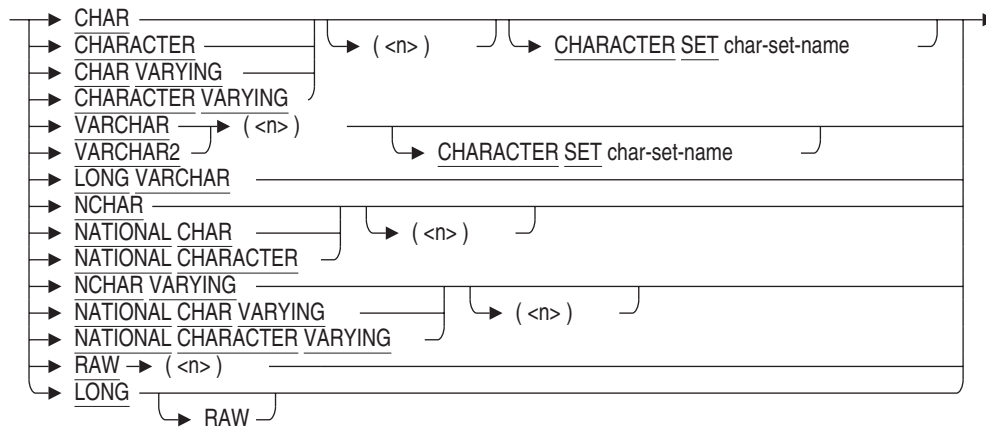


data-type =

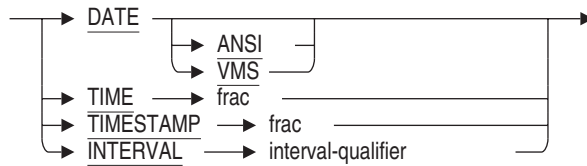


DECLARE LOCAL TEMPORARY TABLE Statement

char-data-types =



date-time-data-types =

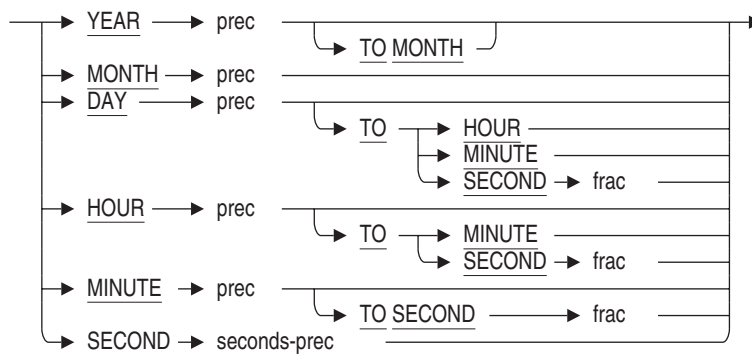


frac =



DECLARE LOCAL TEMPORARY TABLE Statement

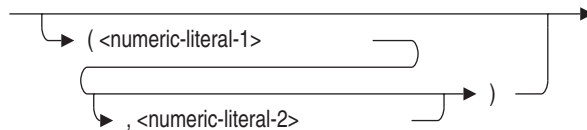
interval-qualifier =



prec =



seconds-prec =



Arguments

dec-local-col-definition

The definition for a column in the table. SQL gives you two ways to specify column definitions:

- By directly specifying a data type to associate with a column name
- By naming a domain that indirectly specifies a data type to associate with a column name

See the CREATE TABLE Statement for more information about column definitions. See Section 2.3 for more information about data types.

DECLARE LOCAL TEMPORARY TABLE Statement

ON COMMIT PRESERVE ROWS

ON COMMIT DELETE ROWS

Specifies whether data is preserved or deleted after a COMMIT statement for declared local temporary tables.

The default, if not specified, is ON COMMIT DELETE ROWS.

table-name

The name of the table you want to declare. You can optionally precede the table-name with an alias-name and a period (.). You must, however, precede the table-name with the keyword MODULE and a period (.), for example, MODULE.EMPL_PAYROLL.

Usage Notes

- You must precede the name of the declared local temporary table with the keyword MODULE and a period (.), for example:

```
SQL> DECLARE LOCAL TEMPORARY TABLE MODULE.empl_payroll
```

```
·  
·  
·
```

- Declared local temporary tables are stored in virtual memory, not in a storage area. They use the same storage segment layout as persistent base tables, but they use additional space in memory for management overhead.

See the *Oracle Rdb Guide to Database Design and Definition* for information on estimating the virtual memory needs of declared local temporary tables.

- Because the metadata is not stored in the database, you cannot use declared local temporary tables in as many places as you use persistent base tables. In particular, declared local temporary tables cannot:
 - Be deleted using the DROP TABLE statement
 - Be modified using the ALTER TABLE statement
 - Be truncated
 - Contain data of the data type LIST OF BYTE VARYING
 - Be referred to in a view or in a storage map
 - Be referred to in a constraint or be defined with a constraint
 - Contain indexes

DECLARE LOCAL TEMPORARY TABLE Statement

- Use triggers
 - Have granted or revoked privileges
 - Be referred to in an interactive or dynamic CREATE OUTLINE statement if the declared local temporary table is outside the definition of a stored module
 - Be referred to in a COMMENT ON statement
 - Be specified in the RESERVING clause of a SET TRANSACTION statement
 - Be displayed using the SHOW statement
 - Be referenced in a COMPUTED BY column of another persistent or declared local temporary table
 - Be exported or imported, unless as part of a module.
- You cannot define column or table constraints in declared local temporary tables. The columns in a declared local temporary table can reference domain constraints.
 - You can use dbkeys with declared local temporary tables.
 - Oracle Rdb does not journal changes to declared local temporary tables but does manage ROLLBACK of changes in a transaction.
 - You can define and write to a declared local temporary table during a read-only transaction.
 - You can qualify the name of the table with an alias name. For example, if the database alias is PERS, the qualified name of PAYCHECK_DECL_TAB is PERS.MODULE.PAYCHECK_DECL_TAB. However, the declared local temporary table name is not an element of a catalog or schema.
 - The following table summarizes the actions you can take using temporary tables and when you can refer to temporary tables.

Action	Types of Temporary Tables		
	Global	Local	Declared Local
Drop table	Yes	Yes	No
Alter table	Yes	No	No
Truncate table	Yes	No	No

DECLARE LOCAL TEMPORARY TABLE Statement

Action	Types of Temporary Tables		
	Global	Local	Declared Local
Add constraints on table or column	Yes	No	No
Refer to table in constraint definition	Yes ²	Yes	No
Refer to domain constraints	Yes	Yes	Yes
Refer to table in storage map	Yes ³	Yes	No
Refer to table in view	Yes	Yes	No
Grant privileges on temporary table	Yes	Yes	No
Refer to table in outline	Yes	Yes	No ¹
Create indexes on table	No	No	No
Use dbkeys on table	Yes	Yes	Yes
Use triggers with table	Yes	No	No
Refer to table in COMMENT ON statement	Yes	Yes	No
Contain LIST OF BYTE VARYING data	No	No	No
Specify in RESERVING clause	Yes ⁴	Yes ⁴	No
Write to table during read-only transaction	Yes	Yes	Yes
Create in a read-only transaction	No	No	Yes
Refer to a table in a computed by column	Yes	Yes	No

¹You can refer to a declared local temporary table if it is defined inside a stored module.

²From a temporary table only.

³Only the ENABLE or DISABLE COMPRESSION attribute may be specified.

⁴Such references are ignored.

For information about global and local temporary tables, see the CREATE TABLE Statement.

- Because the declared local temporary table name is qualified by the keyword MODULE and a period (.), a declared local temporary table can have the same name as a persistent base table or view.

DECLARE LOCAL TEMPORARY TABLE Statement

Examples

Example 1: Declaring and using a declared local temporary table in interactive SQL

```
SQL> DECLARE LOCAL TEMPORARY TABLE MODULE.PAYCHECK_DECL_INT
cont>     (EMPLOYEE_ID ID_DOM,
cont>       LAST_NAME CHAR(14),
cont>       HOURS_WORKED INTEGER,
cont>       HOURLY_SAL   INTEGER(2),
cont>       WEEKLY_PAY   INTEGER(2))
cont>     ON COMMIT PRESERVE ROWS;
SQL> --
SQL> INSERT INTO MODULE.PAYCHECK_DECL_INT
cont>     (EMPLOYEE_ID, LAST_NAME, HOURS_WORKED, HOURLY_SAL, WEEKLY_PAY)
cont>     SELECT P.EMPLOYEE_ID, E.LAST_NAME, P.HOURS_WORKED,
cont>           P.HOURLY_SAL, P.HOURS_WORKED * P.HOURLY_SAL
cont>     FROM EMPLOYEES E, PAYROLL P
cont>     WHERE E.EMPLOYEE_ID = P.EMPLOYEE_ID
cont>           AND P.WEEK_DATE = DATE '1995-08-01';
100 rows inserted

SQL> SELECT * FROM MODULE.PAYCHECK_DECL_INT LIMIT TO 2 ROWS;
EMPLOYEE_ID  LAST_NAME      HOURS_WORKED  HOURLY_SAL      WEEKLY_PAY
00165        Smith          40             30.50            1220.00
00166        Dietrich       40             36.00            1440.00
2 rows selected
```

Example 2: Creating a stored module that contains the following:

- A declared local temporary table, `MODULE.PAYCHECK_DECL_TAB`
- A procedure, `PAYCHECK_INS_DECL`, that inserts weekly salary records into the declared local temporary table, `MODULE.PAYCHECK_DECL_TAB`
- A procedure, `LOW_HOURS_DECL`, that counts the number of employees with less than 40 hours worked

The following example also demonstrates that you can access the declared local temporary table only from within the module.

```
SQL> -- Create the module containing a declared temporary table.
SQL> --
SQL> CREATE MODULE PAYCHECK_DECL_MOD
cont>     LANGUAGE SQL
cont>     DECLARE LOCAL TEMPORARY TABLE MODULE.PAYCHECK_DECL_TAB
cont>     (EMPLOYEE_ID ID_DOM,
cont>     LAST_NAME CHAR(14) ,
cont>     HOURS_WORKED INTEGER, HOURLY_SAL INTEGER(2),
cont>     WEEKLY_PAY   INTEGER(2))
cont>     ON COMMIT PRESERVE ROWS
```

DECLARE LOCAL TEMPORARY TABLE Statement

```
cont> --
cont> -- Create the procedure to insert rows.
cont> --
cont> PROCEDURE PAYCHECK_INS_DECL;
cont> BEGIN
cont>     INSERT INTO MODULE.PAYCHECK_DECL_TAB
cont>         (EMPLOYEE_ID, LAST_NAME, HOURS_WORKED, HOURLY_SAL, WEEKLY_PAY)
cont>         SELECT P.EMPLOYEE_ID, E.LAST_NAME, P.HOURS_WORKED,
cont>             P.HOURLY_SAL, P.HOURS_WORKED * P.HOURLY_SAL
cont>         FROM EMPLOYEES E, PAYROLL P
cont>         WHERE E.EMPLOYEE_ID = P.EMPLOYEE_ID
cont>             AND P.WEEK_DATE = DATE '1995-08-01';
cont> END;
cont> --
cont> -- Create the procedure to count the low hours.
cont> --
cont> PROCEDURE LOW_HOURS_DECL (:cnt INTEGER);
cont> BEGIN
cont>     SELECT COUNT(*) INTO :cnt FROM MODULE.PAYCHECK_DECL_TAB
cont>         WHERE HOURS_WORKED < 40;
cont> END;
cont> END MODULE;
SQL> --
SQL> -- Call the procedure to insert the rows.
SQL> --
SQL> CALL PAYCHECK_INS_DECL();
SQL> --
SQL> -- Declare a variable and call the procedure to count records with
SQL> -- low hours.
SQL> --
SQL> DECLARE :low_hr_cnt integer;
SQL> CALL LOW_HOURS_DECL(:low_hr_cnt);
    LOW_HR_CNT
      2

SQL> --
SQL> -- Because the table is a declared local temporary table, you cannot
SQL> -- access it from outside the stored module that contains it.
SQL> --
SQL> SELECT * FROM MODULE.PAYCHECK_DECL_TAB;
%SQL-F-RELNOTDCL, Table PAYCHECK_DECL_TAB has not been declared in module or
environment
```

DECLARE MODULE Statement

Specifies characteristics, such as character sets, quoting rules, and the default date format for a nonstored module.

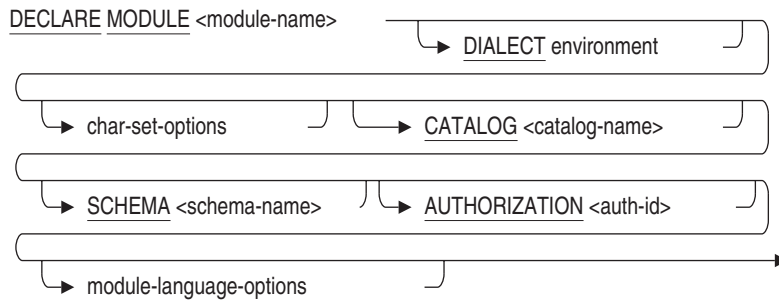
Environment

You can use the DECLARE MODULE statement:

- Embedded in host language programs to be precompiled
- In a context file

This command is not executable.

Format

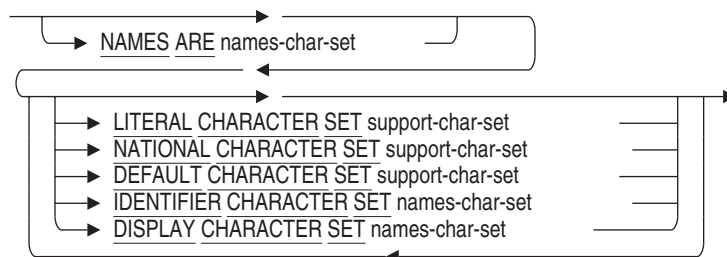


environment =

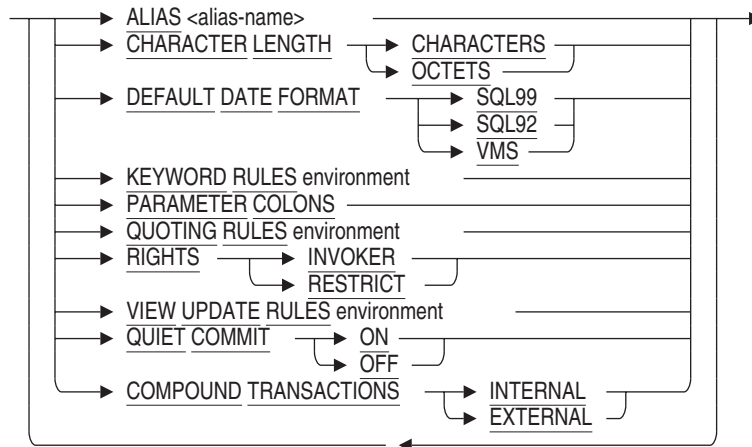


DECLARE MODULE Statement

char-set-options =



module-language-options =



Arguments

ALIAS alias-name

Specifies the module alias. If you do not specify a module alias, the default alias is the authorization identifier for the module.

When the FIPS flagger is enabled, the ALIAS clause (by itself or used with the AUTHORIZATION clause) is flagged as nonstandard syntax.

If the application needs to refer to only one database across multiple modules, it is good practice to use the same alias for the default database in all modules that will be linked to make up an executable image.

DECLARE MODULE Statement

AUTHORIZATION auth-id

Specifies the authorization identifier for the module. If you do not specify a schema clause, the authorization identifier specifies the default schema.

To comply with the ANSI/ISO 1989 standard, specify the **AUTHORIZATION** clause without the schema name. Specify both the **AUTHORIZATION** clause and the schema name to comply with the ANSI/ISO SQL standard.

When you attach to a multischema database, the authorization identifier for each schema is the user name of the user compiling the module. This authorization identifier defines the default alias and schema. You can use the **SCHEMA** clause and the **DECLARE ALIAS** statement to override the defaults.

If you attach to a single-schema database or specify that **MULTISHEMA IS OFF** in your **ATTACH** or **DECLARE ALIAS** statements and you specify both an **AUTHORIZATION** clause and an **ALIAS** clause, the authorization identifier is ignored by SQL unless you use the **RIGHTS RESTRICT** clause. The **RIGHTS RESTRICT** clause causes SQL to use the authorization identifier specified in the module **AUTHORIZATION** clause for privilege checking.

If procedures in the SQL module always qualify table names with an authorization identifier, the **AUTHORIZATION** clause has no effect on SQL statements in the procedures.

When the FIPS flagger is enabled, the omission of an **AUTHORIZATION** clause is flagged as nonstandard ANSI syntax.

CATALOG catalog-name

Specifies the default catalog for the module. **Catalogs** are groups of schemas within a multischema database. If you omit the catalog name when specifying an object in a multischema database, SQL uses the default catalog name **RDB\$CATALOG**. Databases created without the multischema attribute do not have catalogs. You can use the **SET CATALOG** statement to change the current default catalog name in dynamic or interactive SQL.

CHARACTER LENGTH CHARACTERS

CHARACTER LENGTH OCTETS

Specifies whether the length of character string parameters, columns, and domains are interpreted as characters or octets. The default is octets.

DEFAULT CHARACTER SET support-char-set

Specifies the character set for parameters that are not qualified by a character set. The default is **DEC_MCS**. This clause overrides the character set specified in the **NAMES ARE** clause. See Section 2.1 for a list of the allowable character sets.

DECLARE MODULE Statement

DEFAULT DATE FORMAT SQL99

DEFAULT DATE FORMAT VMS

Controls the default interpretation for the data type of the CURRENT_TIMESTAMP built in function and column or CAST expressions with the DATE data type. The DATE and CURRENT_TIMESTAMP data types can be either VMS or ANSI/ISO Standard format.

If you specify VMS, both data types are interpreted as VMS format. The VMS format DATE and CURRENT_TIMESTAMP contain YEAR TO SECOND fields.

If you specify SQL99 or SQL92, both data types are interpreted as SQL standard format. The SQL format DATE contains only the YEAR TO DAY fields.

The default is VMS.

Use the DEFAULT DATE FORMAT clause, rather than the SQLOPTIONS = ANSI_DATE qualifier because the qualifier will be deprecated in a future release.

DIALECT

Controls the following settings:

- Whether the length of character string parameters, columns, and domains are interpreted as characters or octets
- Whether double quotation marks are interpreted as string literals or delimited identifiers
- Whether or not identifiers can be keywords
- Which views are read-only
- Whether columns with the DATE or CURRENT_TIMESTAMP data type are interpreted as VMS or SQL99 format

The DIALECT clause lets you specify the settings with one clause, instead of specifying each setting individually. Because the module processor processes the module clauses sequentially, the DIALECT clause can override the settings of clauses specified before it or be overridden by clauses specified after it.

The following statements are specific to the SQL99 dialect:

- The default constraint evaluation time setting changes from DEFERRABLE to NOT DEFERRABLE.
- Conversions between character data types when storing data or retrieving data will raise exceptions or warnings in certain situations.

DECLARE MODULE Statement

- You can specify DECIMAL or NUMERIC for formal parameters in SQL modules, and declare host language parameters with packed decimal or signed numeric storage format. SQL generates an error message if you attempt to exceed the precision specified.
- The USER keyword specifies the current active user name for a request.
- A warning is generated when a NULL value is eliminated from a SET function.
- The WITH CHECK OPTION clause on views returns a discrete error code from an integrity constraint failure.
- An exception is generated with non-null terminated C strings.

Table 8-5 shows the dialect settings for each environment.

DISPLAY CHARACTER SET *names-char-set*

Specifies the character set encoding and characteristics expected of text strings returned back to SQL from Oracle Rdb. See the Usage Notes under CREATE DATABASE Statement for additional information.

IDENTIFIER CHARACTER SET *names-char-set*

Specifies the character set used for database object names such as table names and column names. This clause overrides the character set specified in the NAMES ARE clause. See Section 2.1.5 for a list of allowable character sets and option values.

The specified character set must contain ASCII characters.

KEYWORD RULES

Controls whether or not identifiers can be keywords. If you specify SQL99, SQL92, SQL89, or MIA, you cannot use keywords as identifiers, unless you enclose them in double quotation marks. If you specify SQLV40, you can use keywords as identifiers. The default is SQLV40.

Use the KEYWORD RULES clause, rather than the SQLOPTIONS = ANSI_IDENTIFIER qualifier because the qualifier will be deprecated in a future release.

LITERAL CHARACTER SET *support-char-set*

Specifies the character set for literals that are not qualified by a character set or national character set. If you do not specify a character set in this clause or in the NAMES ARE clause, the default is DEC_MCS. This clause overrides the character set for unqualified literals specified in the NAMES ARE clause. See Section 2.1 for a list of the allowable character sets.

DECLARE MODULE Statement

MODULE module-name

An optional name for the nonstored module. If you do not supply a module name, the default name is `SQL_MODULE`.

Use any valid OpenVMS name. (See Section 2.2 for more information on user-supplied names.) However, the name must be unique among the modules that are linked together to form an executable image.

NAMES ARE names-char-set

Specifies the character set used for the default, identifier, and literal character sets for the module. Also specifies the character string parameters that are not qualified by a character set or national character set. If you do not specify a character set, the default is `DEC_MCS`.

You must ensure that the character set specified in this clause matches the character set of all the databases attached to by any particular connection and must contain ASCII characters. See Section 2.1.5 for a list of the allowable character sets.

NATIONAL CHARACTER SET support-char-set

Specifies the character set for literals qualified by the national character set. See Section 2.1 for a list of the allowable character sets.

PARAMETER COLONS

If you use the `PARAMETER COLONS` clause, all parameter names must begin with a colon (:). This is valid in context files for module language only. This rule applies to both declarations and references of module language procedure parameters. If you do not use this clause, no parameter name can begin with a colon.

The current default behavior is no colons are used. However, this default is deprecated syntax. In the future, required colons will be the default because it allows processing of ANSI/ISO SQL standard modules.

Use the `PARAMETER COLONS` clause, rather than the `SQLOPTIONS = ANSI_PARAMETERS` qualifier because the qualifier will be deprecated in a future release.

QUOTING RULES

Controls whether double quotation marks are interpreted as string literals or delimited identifiers. If you specify `SQLV40`, SQL interprets double quotation marks as literals. All other dialects interpret double quotation marks as delimited identifiers. The default is `SQLV40`.

Use the `QUOTING RULES` clause, rather than the `SQLOPTIONS = ANSI_QUOTING` qualifier because the qualifier will be deprecated in a future release.

DECLARE MODULE Statement

RIGHTS INVOKER RIGHTS RESTRICT

Specifies whether or not a module must be executed by a user whose authorization identifier matches the module authorization identifier.

If you specify **RESTRICT**, SQL bases privilege checking on the default authorization identifier. The default authorization identifier is the authorization identifier of the user who compiles a module, unless you specify a different authorization identifier using an **AUTHORIZATION** clause in the module. The **RESTRICT** option causes SQL to compare the user name of the person who executes a module with the default authorization identifier and prevents any user other than one with the correct authorization identifier from invoking that module. All applications that use multischema restrict the invoker by default.

If you specify **INVOKER**, SQL bases the privilege on the authorization identifier of the user running the module. The default is **INVOKER**.

Use the **RIGHTS** clause, rather than the **SQLOPTIONS = ANSI_ AUTHORIZATION** qualifier because the qualifier will be deprecated in a future release.

SCHEMA schema-name

Specifies the default schema name for the module. The **default schema** is the schema to which SQL statements refer if those statements do not qualify table names and other schema names with an authorization identifier. If you do not specify a default schema name for a module, you must specify a default authorization identifier.

Using the **SCHEMA** clause, separate modules can each declare different schemas as default schemas. This can be convenient for an application that needs to refer to more than one schema. By putting SQL statements that refer to a schema in the appropriate module's procedures, you can minimize tedious qualification of schema element names in those statements.

When you specify **SCHEMA schema-name AUTHORIZATION auth-id**, you specify the schema name and the schema authorization identifier for the module. The schema authorization identifier is considered the owner and creator of the schema and everything in it.

VIEW UPDATE RULES

Specifies whether or not the SQL module processor applies the ANSI/ISO SQL standard for updatable views to all views created during compilation.

DECLARE MODULE Statement

If you specify SQL99, SQL92, SQL89, or MIA, the SQL module processor applies that ANSI/ISO SQL standard for updatable views to all views created during compilation. Views that do not comply with the specified ANSI/ISO SQL standard for updatable views cannot be updated.

The specified ANSI/ISO standard for updatable views requires the following conditions to be met in the SELECT statement:

- The DISTINCT keyword is not specified.
- Only column names can appear in the select list. Each column name can appear only once. Functions and expressions such as max(column_name) or column_name +1 cannot appear in the select list.
- The FROM clause refers to only one table. This table must be either a base table or a derived table that can be updated.
- The WHERE clause does not contain a subquery.
- The GROUP BY clause is not specified.
- The HAVING clause is not specified.

If you specify SQLV40, SQL does not apply the ANSI/ISO standard for updatable views. Instead, SQL considers views that meet the following conditions to be updatable:

- The DISTINCT keyword is not specified.
- The FROM clause refers to only one table. This table must be either a base table or a derived table that can be updated.
- The WHERE clause does not contain a subquery.
- The GROUP BY clause is not specified.
- The HAVING clause is not specified.

Example

Example 1: Declaring a module specifying character strings of different character sets

Assuming that the character sets for the database match the character sets specified in the program, the following example shows a simple SQL precompiled C program that retrieves one row from the COLOURS table.

DECLARE MODULE Statement

```
/* This SQL precompiled program does some simple tests of character length
 * and character sets.
 */
#include stdio
#include descrip

main()
{
/* Specify CHARACTER LENGTH CHARACTERS in the DECLARE MODULE statement.
 * In addition, specify the NAMES, NATIONAL, and DEFAULT character sets.
 */
EXEC SQL DECLARE MODULE CCC_COLOURS
        NAMES ARE DEC_KANJI
        NATIONAL CHARACTER SET KANJI
        SCHEMA RDB$SCHEMA
        AUTHORIZATION SQL_SAMPLE
        CHARACTER LENGTH CHARACTERS
        DEFAULT CHARACTER SET DEC_KANJI
        ALIAS RDB$DBHANDLE;

/* If you do not specify character sets in the DECLARE ALIAS statement, SQL
 * uses the character sets of the compile-time database.
 */
EXEC SQL DECLARE ALIAS FILENAME MIA_CHAR_SET;

int      SQLCODE;

/* Because the default character set is DEC_KANJI, you do not need to qualify
 * the variable dec_kanji_p with the character set, but you must declare
 * char in lowercase.
 */
char  dec_kanji_p[31];

/* When you declare a parameter with lowercase char, SQL considers the
 * character set unspecified and allocates single-octet characters.
 */
char  english_p[31];

/* When you specify the character set, SQL allocates single- or multi-octet
 * characters, depending upon the character set.
 */
char  CHARACTER SET DEC_MCS  french_p[31];
char  CHARACTER SET KANJI   japanese_p[31];
      .
      .
      .

/* Select one row from the COLOURS table. */
EXEC SQL SELECT ENGLISH, FRENCH, JAPANESE, ROMAJI,
        KATAKANA, HINDI, GREEK, ARABIC, RUSSIAN
        INTO :english_p, :french_p, :japanese_p, :dec_kanji_p,
        :katakana_p, :devanagari_p, :isolatingreek_p,
        :isolatinarabic_p, :isolatincyrillic_p
        FROM COLOURS LIMIT TO 1 ROW;
```

DECLARE MODULE Statement

```
        if (SQLCODE != 0)
            SQL$SIGNAL();

    printf ("\nENGLISH: %s", english_p);
    printf ("\nFRENCH: %s", french_p);
    printf ("\nJAPANESE: %s", japanese_p);
    printf ("\nROMAJI: %s", dec_kanji_p);
    printf ("\nKATAKANA: %s", katakana_p);
    printf ("\nHINDI: %s", devanagari_p);
    printf ("\nGREEK: %s", isolatingreek_p);
    printf ("\nARABIC: %s", isolatinarabic_p);
    printf ("\nRUSSIAN: %s", isolatincyrillic_p);

EXEC SQL    ROLLBACK;
}
```


DECLARE PROCEDURE Statement

DECLARE PROCEDURE Statement

Declares a procedure interface for use in database definition statements.

The `DECLARE PROCEDURE` statement is documented under the `DECLARE Routine Statement`. For complete information on declaring a procedure, see the `DECLARE Routine Statement`.

DECLARE Routine Statement

DECLARE Routine Statement

Declares a routine interface for use in database definition statements. A routine is either a function or a procedure.

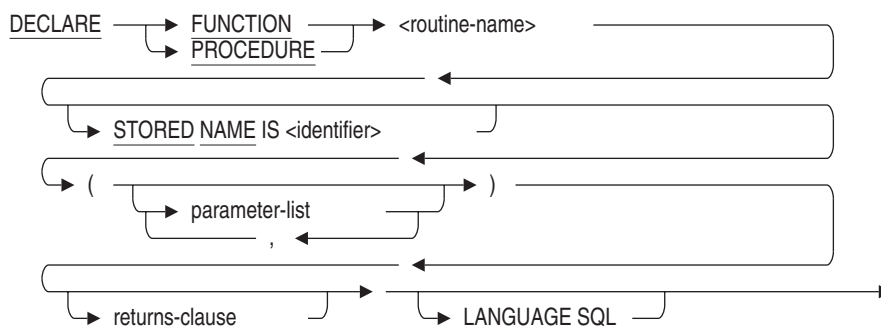
The declared routine acts as a template for calls to the function or procedure in DDL statements such as `CREATE TABLE`, `CREATE VIEW` and `CREATE MODULE`. The template allows Rdb to validate that the routine is correctly named, is passed the correct number of parameters and that those parameters are passed compatible arguments. For functions the returned data type is used to calculate data types for `COMPUTED BY`, `AUTOMATIC` and other stored value expressions.

Environment

You can use the `DECLARE Routine` statement:

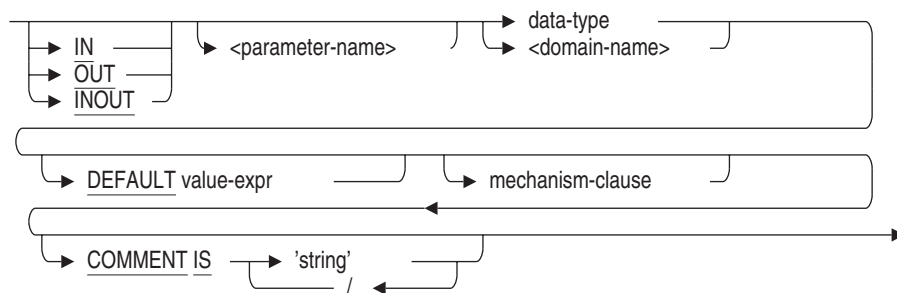
- In interactive SQL
- In dynamic SQL as a statement to be dynamically executed

Format

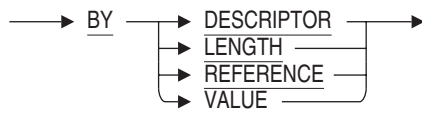


DECLARE Routine Statement

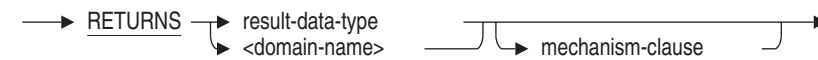
parameter-list =



mechanism-clause =



returns-clause =



Arguments

DEFAULT value-expr

Specifies the default value of a parameter for a function or procedure defined with mode IN. If you omit this parameter or if the CALL statement argument list or function invocation specifies the DEFAULT keyword, then the value-expr specified with this clause is used. The parameter uses NULL as the default if you do not specify a value expression explicitly.

FUNCTION

Declares a function definition.

A function optionally accepts a list of IN parameters, always returns a value, and is referenced by name as an element of a value expression.

LANGUAGE SQL

Names the language that calls the routine.

DECLARE Routine Statement

mechanism-clause

Defines the passing mechanism for an external routine. The following list describes the passing mechanisms.

- **BY DESCRIPTOR**
Allows passing character data with any parameter access mode to routines compiled by language compilers that implement the OpenVMS calling standard.
- **BY LENGTH**
The LENGTH passing mechanism is the same as the DESCRIPTOR passing mechanism.
- **BY REFERENCE**
Allows passing data with any parameter access mode as a reference to the actual data.

This is the default passing mechanism for parameters. This is also the default passing mechanism for a function value returning character data.
- **BY VALUE**
Allows passing data with the IN parameter access mode to a routine as a value and allows functions to return a value.

This is the default passing mechanism for a function value returning noncharacter data.

parameter-list

The optional parameters of the routine. For each parameter you can specify a parameter access mode (IN, OUT, and INOUT), a parameter name, a data type, and a passing mechanism (by DESCRIPTOR, LENGTH, REFERENCE, or VALUE).

The parameter access mode (IN, OUT, and INOUT) is optional and specifies how the parameter is accessed (whether it is read, written, or both). IN signifies read only, OUT signifies write only, and INOUT signifies read and write. The parameter access mode defaults to IN.

Only the IN parameter access mode may be specified with parameters to a function. Any of the parameter access modes (IN, OUT, and INOUT) may be specified with parameters to a procedure.

The parameter name is prefixed with a colon (:). The parameter name must be unique within the routine parameters.

The data type is required and describes the type of parameter using either an SQL data type or a domain name.

DECLARE Routine Statement

You cannot declare a parameter as the LIST OF BYTE VARYING data type.

PROCEDURE

Declares a procedure definition.

A procedure optionally accepts a list of IN, OUT, or INOUT parameters, and is referenced by name in a CALL statement.

RETURNS result-data-type

RETURNS domain-name

Describes a function (returned) value. You can specify a data type and a passing mechanism (BY DESCRIPTOR, LENGTH, REFERENCE, or VALUE). The function value is, by definition, an OUT access mode value.

The data type is required and describes the type of parameter using either an SQL data type or a domain name.

You cannot declare a function value as the LIST OF BYTE VARYING data type.

routine-name

The name of the external routine. The name must be unique among external and stored routines in the schema and can be qualified with an alias or, in a multischema database, a schema name.

STORED NAME IS identifier

The name that Oracle Rdb uses to access the routine when defined in a multischema database. The stored name allows you to access multischema definitions using interfaces that do not recognize multiple schemas in one database. You cannot specify a stored name for a routine in a database that does not allow multiple schemas. For more information about stored names, see Section 2.2.18.

Usage Notes

- If an additional DECLARE statement is executed with the same routine name then it must be identical to the existing definition.
- The routine that is created using CREATE FUNCTION, CREATE PROCEDURE, or CREATE MODULE statements must match exactly the number of parameters, the data types (domains can be replaced with the base data types or vice versa), passing mechanism (BY VALUE, BY REFERENCE, BY LENGTH, BY DESCRIPTOR), and mode (IN, OUT and INOUT).

DECLARE Routine Statement

- The `DEFAULT` clause on parameters must be specified so that the minimum and maximum parameter counts can be calculated for the routine. However, this `DEFAULT` value is not used and may be specified as `NULL`, i.e. a placeholder.
- A declared routine remains part of the session until it is replaced by a `CREATE FUNCTION`, `CREATE PROCEDURE`, or `CREATE MODULE` statement.

If a `CREATE FUNCTION`, `CREATE PROCEDURE`, or `CREATE MODULE` statement is rolled back then any declared routine it replaced is also eliminated. Therefore, a new `DECLARE` will be required in such cases.

- If the session is disconnected before a `CREATE` statement has defined the true routine body (stored or external) then attempts to use the database objects which reference those routines will fail.

This is similar to the behavior observed after using `DROP ... CASCADE`. i.e. there are unresolved references which must be corrected by creating those objects.

- Tools such as `SQL EXPORT` and `IMPORT` and `RMU Extract` use the `DECLARE` routine facility to allow forward references in generated database definition operations.

For `RMU Extract` the `/ITEM=FORWARD_REFERENCES` qualifier must be used to enable the output of the `DECLARE` statements. For `SQL EXPORT` this is the default setting which can be disabled using the `NO FORWARD_REFERENCES` clause with the `EXPORT` or `IMPORT` commands.

Examples

Example 1: Defining a domain and referencing an external function

```
SQL> create domain MONEY as integer (2);
SQL>
SQL> create function INTEREST_PAID
cont>   (in :amt MONEY)
cont>   returns MONEY;
cont>   external
cont>     language C
cont>     parameter style GENERAL;
SQL>
SQL> alter domain MONEY
cont>   add
cont>     check (INTEREST_PAID (value) > 0)
cont>     not deferrable;
```

DECLARE Routine Statement

Once the ALTER DOMAIN is completed, neither the function nor the domain can be defined before the other. Here is a fragment of the result of executing the output from the RMU Extract command.

```
SQL> create domain MONEY
cont>     INTEGER (2)
cont>     check((INTEREST_PAID(value) > 0))
cont>     not deferrable;
%SQL-F-RTNNOTDEF, function or procedure INTEREST_PAID is not defined
SQL>
SQL> commit work;
SQL> create function INTEREST_PAID (
cont>     in     :AMT
cont>           MONEY
cont>           by reference)
cont>     returns
cont>           MONEY by value
cont>     language SQL;
cont>     external
cont>           language C
cont>           parameter style GENERAL
cont>     deterministic
cont>     called on null input
cont>     ;
%SQL-F-NO_SUCH_FIELD, Domain MONEY does not exist in this database or schema
SQL> commit work;
```

This problem is avoided for RMU Extract by adding the FORWARD_ REFERENCES item to the command line:

```
$ RMU/EXTRACT/ITEM=(ALL, FORWARD_REFERENCES) databasename/OUTPUT=script.SQL
```

The script now contains a forward declaration of the function INTEREST_PAID so that execution of the script can succeed.

DECLARE Routine Statement

```
SQL> declare function INTEREST_PAID (  
cont>     in      :AMT  
cont>         INTEGER (2))  
cont>     returns  
cont>         INTEGER (2)  
cont>     ;  
SQL>  
SQL> create domain MONEY  
cont>     INTEGER (2)  
cont>     check((INTEREST_PAID(value) > 0))  
cont>     not deferrable;  
SQL>  
SQL> commit work;  
SQL> create function INTEREST_PAID (  
cont>     in      :AMT  
cont>         MONEY  
cont>         by reference)  
cont>     returns  
cont>         MONEY by value  
cont>     language SQL;  
cont>     external  
cont>         language C  
cont>         parameter style GENERAL  
cont>     deterministic  
cont>     called on null input  
cont>     ;  
SQL> commit work;
```

DECLARE STATEMENT Statement

Documents a statement name later used in a PREPARE statement in dynamic SQL. SQL does not require DECLARE STATEMENT statements and does not generate any code when it precompiles them. They are entirely optional.

Environment

You can issue the DECLARE STATEMENT statement only in host language programs to be precompiled.

Format

DECLARE → <statement-name> → STATEMENT
 , ←

Arguments

statement-name STATEMENT

Specifies the name of a statement later referred to in one of the following embedded dynamic statements:

- PREPARE
- DECLARE CURSOR
- DESCRIBE

Example

Example 1: Declaring a statement name in a PL/I program

This example shows a program line that declares a statement name DYNAMIC_STATEMENT. Later lines in the example show how DECLARE CURSOR, PREPARE, and DESCRIBE statements refer to it. Because you do not have to declare a statement explicitly, the DECLARE STATEMENT statement is always optional.

DECLARE STATEMENT Statement

```
EXEC SQL DECLARE DYNAMIC_STATEMENT STATEMENT;
/* Declare the SQL Communications Area. */
EXEC SQL INCLUDE SQLCA;
/* Declare the SQL Descriptor Area. */
EXEC SQL INCLUDE SQLDA;

/* The program declares the host language variable
   STATEMENT_STRING and stores in it the
   character string containing a SELECT
   statement to be executed dynamically. */
      .
      .
      .
EXEC SQL DECLARE CURSOR1 CURSOR FOR DYNAMIC_STATEMENT;
EXEC SQL PREPARE OBJECT_STATEMENT FROM STATEMENT_STRING;
EXEC SQL DESCRIBE OBJECT_STATEMENT INTO SQLDA;

/* The program sets up pointers in the
   SQLDATA field of the SQLDA to the data
   area (host language variables or dynamic
   memory, for example) to receive the data
   from the cursor. */
      .
      .
      .
EXEC SQL OPEN CURSOR1;
DO WHILE (SQLCODE = 0);
      EXEC SQL FETCH CURSOR1 USING DESCRIPTOR SQLDA;

/* The program prints or otherwise
   processes rows of the result tables. */
      .
      .
      .
END;

EXEC SQL CLOSE CURSOR1;
```

DECLARE TABLE Statement

Explicitly declares a table or view definition in a program. For tables named in a DECLARE TABLE statement, SQL does not check the schema to compare the definition with the explicit declaration.

An explicit table declaration is useful to:

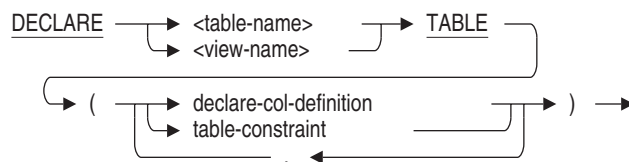
- Document the definition in the source code of the program
- Allow references to tables that do not exist when SQL precompiles the program, including:
 - Tables created in other modules of the program
 - Tables created dynamically
- Improve precompiler performance because SQL does not need to attach to the schema to retrieve the table definition
- Make it easier to check that the declaration correctly corresponds to a host structure the program uses to hold values from or for the table
- Declare only a subset of columns contained in the schema definition of the table if the program needs to use only some of the columns

Environment

You can use the DECLARE TABLE statement:

- Embedded in host language programs to be precompiled
- In a context file
- As part of the DECLARE section in an SQL module

Format

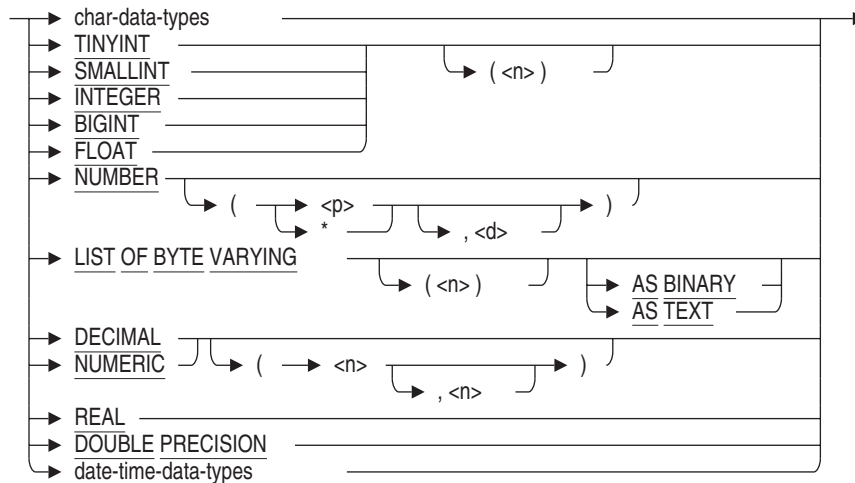


DECLARE TABLE Statement

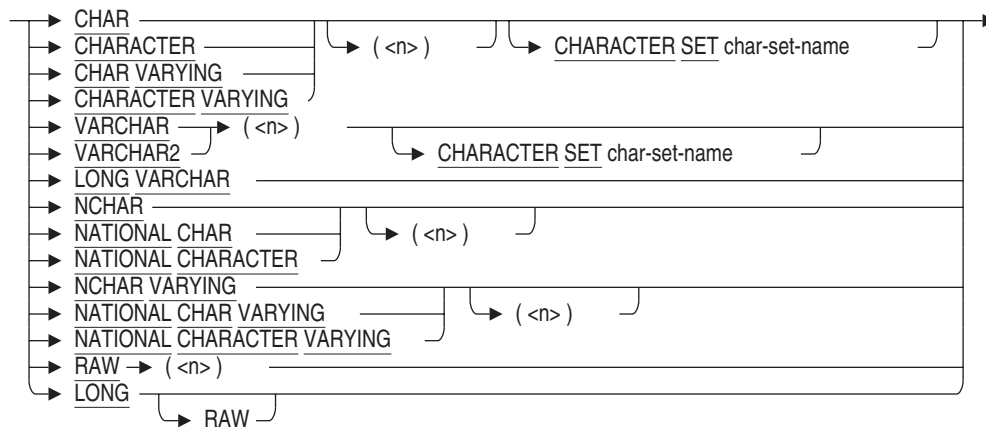
declare-col-definition =



data-type =

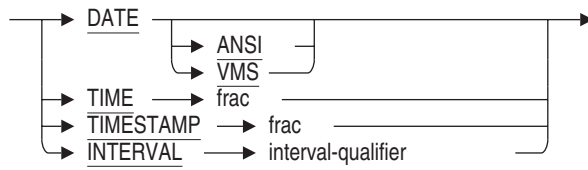


char-data-types =

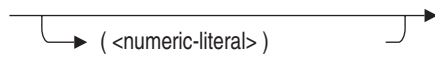


DECLARE TABLE Statement

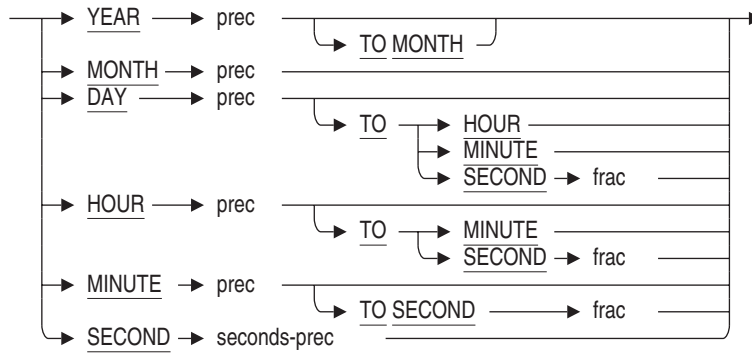
date-time-data-types =



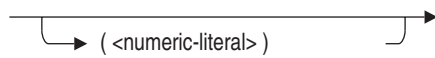
frac =



interval-qualifier =

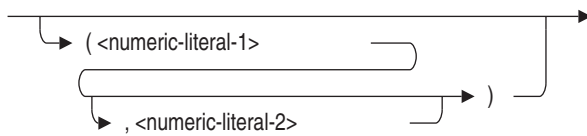


prec =

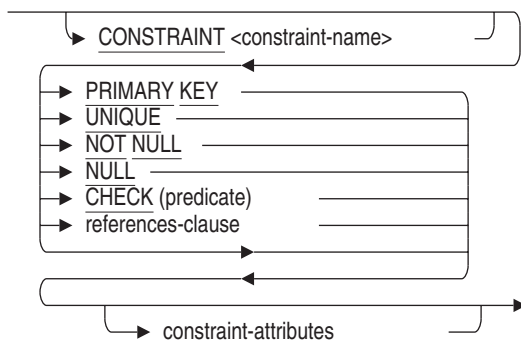


DECLARE TABLE Statement

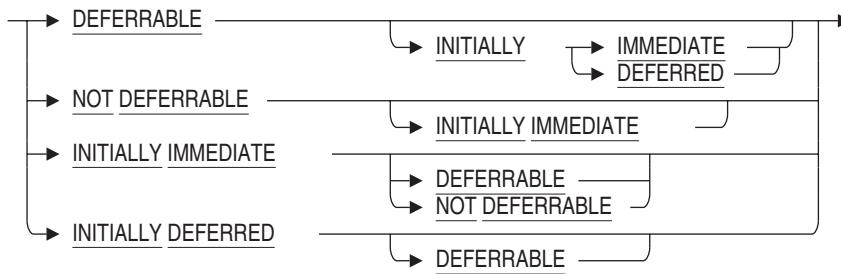
seconds-prec =



col-constraint=



constraint-attributes =



DECLARE TABLE Statement

sql-and-dtr-clause =

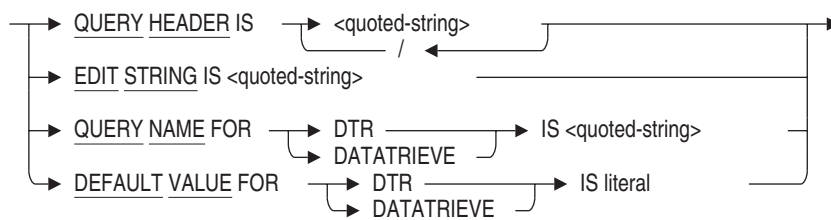


table-constraint =

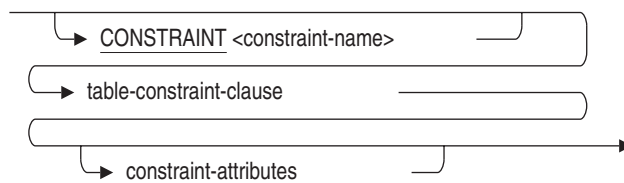
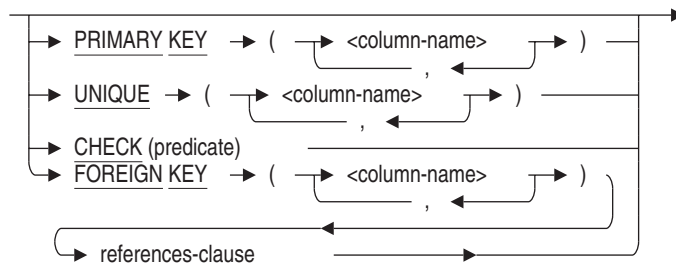


table-constraint-clause =



Arguments

character-set-name

A valid character set name. See Section 2.1 for more information on character sets.

col-constraint

A column constraint. See the CREATE TABLE Statement for more information about column constraints.

DECLARE TABLE Statement

column-name

The name of the column you want to define.

data-type

The data type of the column you want to define. See Section 2.3 for more information on data types.

date-time-data-types

Data types for dates, times, and intervals. See Section 2.3.2 for more information on date-time data types.

declare-col-definition

The definition for a column in the table. The column definition must correspond to the table definition in the schema.

See the CREATE TABLE Statement for more information about column definitions.

However, you cannot refer to domain names in a DECLARE TABLE statement. For tables whose definitions refer to domain names, you must substitute the data type and size of the domain for the domain name.

frac

interval-qualifier

prec

seconds-prec

Precision specifications for date-time data types. See Section 2.3.2 for more information.

references-clause

See the CREATE TABLE Statement for more information.

sql-and-dtr-clause

Optional SQL and DATATRIEVE formatting clause. See Section 2.5 for more information about formatting clauses.

table-name

view-name

The name of the table or view definition you want to declare.

table-constraint

A constraint definition that applies to the whole table. See the CREATE TABLE Statement for more information about specifying table constraints.

DECLARE TABLE Statement

Usage Notes

SQL uses the declaration in the DECLARE TABLE statement when it precompiles embedded SQL statements or processes the module procedures that refer to the table. Therefore, the columns in the declaration should match the columns in the schema definition. However, the table or view definition to which the declaration in the DECLARE TABLE statement corresponds does not have to exist before a program can issue a DECLARE TABLE statement. The program can create the table after it declares it.

Examples

Example 1: Declaring the table EMPLOYEES in a COBOL program

```
EXEC SQL
DECLARE EMPLOYEES TABLE
  (EMPLOYEE_ID                CHAR (5)
   CONSTRAINT EMP_EMPLOYEE_ID_NOT_NULL
   NOT NULL,
   LAST_NAME                  CHAR (14),
   FIRST_NAME                 CHAR (10),
   MIDDLE_INITIAL             CHAR (1),
   ADDRESS_DATA_1             CHAR (25),
   ADDRESS_DATA_2             CHAR (25),
   CITY                       CHAR (20),
   STATE                      CHAR (2),
   POSTAL_CODE                CHAR (5),
   SEX                        CHAR (1),
   CONSTRAINT EMP_SEX_VALUES
   CHECK (
     SEX IN ('M', 'F') OR SEX IS NULL
   ),
   BIRTHDAY                   DATE ,
   STATUS_CODE                CHAR (1)
   CONSTRAINT EMP_STATUS_CODE_VALUES
   CHECK (
     STATUS_CODE IN ('0', '1', '2')
     OR STATUS_CODE IS NULL
   )
)
END_EXEC
```

DECLARE TRANSACTION Statement

DECLARE TRANSACTION Statement

Specifies the characteristics for a default transaction. A **transaction** is a group of statements whose changes can be made permanent or undone only as a unit.

A transaction ends with a COMMIT or ROLLBACK statement. If you end the transaction with the COMMIT statement, all changes made to the database by the statements are made permanent. If you end the transaction with the ROLLBACK statement, the statements do not take effect.

The characteristics specified in a DECLARE TRANSACTION statement affect all transactions (except those started by the SET TRANSACTION or START TRANSACTION statement) until you issue another DECLARE TRANSACTION statement. The characteristics specified in a SET TRANSACTION or START TRANSACTION statement affect only that transaction.

A DECLARE TRANSACTION statement does not start a transaction. The declarations made in a DECLARE TRANSACTION statement do not take effect until SQL starts a new transaction. SQL starts a new transaction with the first executable data manipulation or data definition statement following a DECLARE TRANSACTION, COMMIT, or ROLLBACK statement. In the latter case (following a COMMIT or ROLLBACK statement), SQL applies the transaction characteristics you declared for the transaction that just ended to the next one you start.

In addition to the DECLARE TRANSACTION statement, you can specify the characteristics of a transaction in one of two ways:

- If you specify the SET TRANSACTION or START TRANSACTION statement, the declarations in the statement take effect immediately and SQL starts a new transaction.
- You can retrieve and update data without declaring or setting a transaction explicitly. If you omit the DECLARE TRANSACTION, SET TRANSACTION or START TRANSACTION statements, SQL automatically starts a transaction (using the read/write option) with the first executable data manipulation or data definition statement following a COMMIT or ROLLBACK statement.

See the Usage Notes for examples of when you would want to use the DECLARE TRANSACTION statement instead of the SET TRANSACTION or START TRANSACTION statement.

DECLARE TRANSACTION Statement

You can specify many options with the DECLARE TRANSACTION statement, including:

- A transaction mode (READ ONLY/READ WRITE/BATCH UPDATE)
- A lock specification clause (RESERVING options)
- A wait mode (WAIT/NOWAIT)
- An isolation level
- A constraint evaluation specification clause
- Multiple sets of all the preceding options for each database involved in the transaction (ON clause)

Environment

You can use the DECLARE TRANSACTION statement:

- In interactive SQL
- Embedded in host language programs to be precompiled
- In a context file
- As part of the DECLARE section in an SQL module
- As part of the module header in a CREATE MODULE statement
- In dynamic SQL as a statement to be dynamically executed

In host language programs, you can have only a single DECLARE TRANSACTION statement in each separately compiled source file. See the Usage Notes for more information.

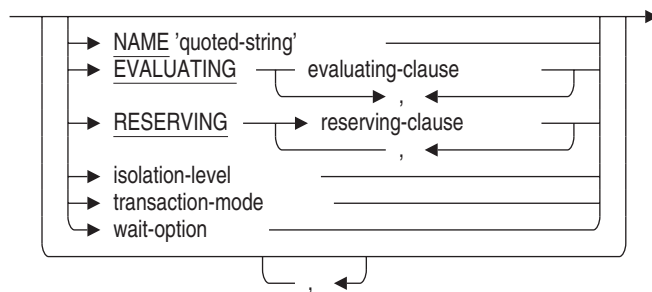
The DECLARE TRANSACTION statement is an extension to standard SQL syntax. If your program must adhere to standard SQL syntax, you can isolate a DECLARE TRANSACTION statement by putting it in a context file. For more information on context files, see the *Oracle Rdb Guide to SQL Programming*.

Format

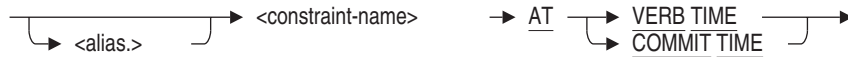


DECLARE TRANSACTION Statement

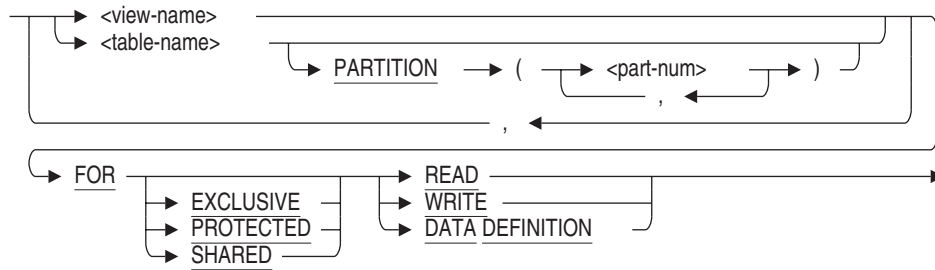
tx-options =



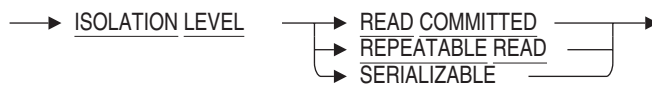
evaluating-clause =



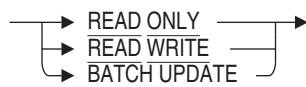
reserving-clause =



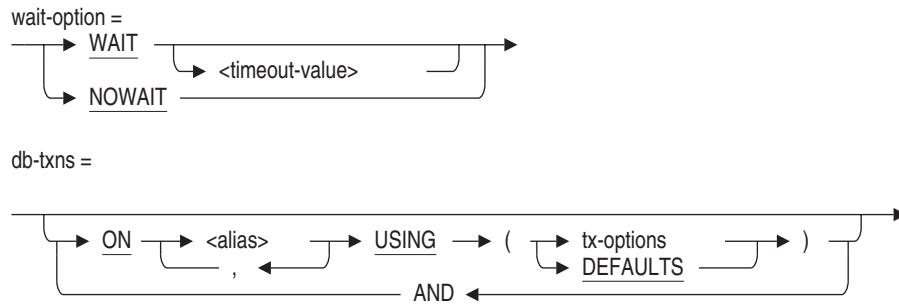
isolation-level =



transaction-mode =



DECLARE TRANSACTION Statement



Arguments

The `DECLARE TRANSACTION` arguments are the same as the arguments for the `SET TRANSACTION` statement. See the `SET TRANSACTION` Statement for more information about the arguments for both statements.

Defaults

The `DECLARE TRANSACTION` defaults are the same as the defaults for the `SET TRANSACTION` statement. See the `SET TRANSACTION` Statement for complete information.

In general, you should not rely on default transaction characteristics. Use explicit `DECLARE TRANSACTION` statements, specifying read/write, read-only, or batch-update options; a list of tables in the `RESERVING` clause; and a share mode and lock type for each table. The more specific you are in a `DECLARE TRANSACTION` statement, the more efficient your database operations will be.

When a transaction starts using characteristics specified in a `DECLARE TRANSACTION` statement, any transaction characteristics unspecified in the `DECLARE TRANSACTION` statement take the SQL defaults. This is true even if the characteristics unspecified in `DECLARE TRANSACTION` were specified in an earlier `SET` or `DECLARE TRANSACTION` statement.

DECLARE TRANSACTION Statement

Usage Notes

The following notes are particular to DECLARE TRANSACTION. See the SET TRANSACTION Statement for usage notes that are common to both DECLARE TRANSACTION and SET TRANSACTION statements.

- The DECLARE TRANSACTION statement is not executable, and therefore, does not start a transaction. (The declarations in a DECLARE TRANSACTION statement take effect when SQL starts a new transaction; that is, with the first executable data manipulation or data definition statement following the DECLARE TRANSACTION, COMMIT, or ROLLBACK statement.)

You can apply only one DECLARE TRANSACTION statement to a host language source file or to an SQL module. Use the SET TRANSACTION statement to change transaction characteristics in programs that were first specified using the DECLARE TRANSACTION statement.

The following are advantages offered by the DECLARE TRANSACTION statement:

- It can establish transaction defaults for an interactive SQL session, a module or single host language file in a program, or any statements executed dynamically from a module. You might, for example, specify DECLARE TRANSACTION READ ONLY in the SQLINI.SQL file you create to set up your interactive SQL environment.

In interactive SQL, the characteristics specified by a DECLARE TRANSACTION statement are valid until you enter another DECLARE TRANSACTION statement. (A COMMIT or ROLLBACK statement followed by a SET TRANSACTION statement may start a transaction with different characteristics, but subsequent transactions started implicitly will have the characteristics specified in the last DECLARE TRANSACTION statement.)

If you specify characteristics using a SET TRANSACTION statement, however, the characteristics apply only to that transaction. You must reenter the statement after every COMMIT or ROLLBACK statement to establish those characteristics again.

The following sequence shows a DECLARE TRANSACTION statement followed by a SET TRANSACTION statement. Note that the SET TRANSACTION statement is followed by a ROLLBACK statement:

DECLARE TRANSACTION Statement

```
SQL> -- Declares default characteristics for transactions:
SQL> --
SQL> DECLARE TRANSACTION READ WRITE;
SQL> --
SQL> -- There is no transaction started; can start
SQL> -- transaction with characteristics different
SQL> -- from the declared characteristics using a
SQL> -- SET TRANSACTION statement:
SQL> --
SQL> SET TRANSACTION READ ONLY;
SQL> --
SQL> -- Roll back the transaction started by
SQL> -- the SET TRANSACTION statement:
SQL> --
SQL> ROLLBACK;
SQL> --
SQL> -- The default transaction characteristics are still those
SQL> -- specified in the DECLARE TRANSACTION statement, and
SQL> -- apply to the transaction started when this SELECT
SQL> -- statement executes:
SQL> --
SQL> SELECT * FROM EMPLOYEES;
```

- You can include the DECLARE TRANSACTION statement in an SQL context file.

The section in the *Oracle Rdb Guide to SQL Programming* about program transportability explains when you may need an SQL context file to support a program that includes SQL statements.

- In contrast to the DECLARE TRANSACTION statement, SET TRANSACTION is an executable statement that specifies and starts one transaction. You can include multiple SET TRANSACTION statements in a host language source file or in an SQL language module. The SET TRANSACTION statement has the following advantages:
 - It gives you explicit control over when transactions are started.
 - It provides flexibility for changing transaction characteristics in a program source file.
- In precompiled programs, you can have only a single DECLARE TRANSACTION statement in each separately compiled source file. It must precede any executable SQL statement and follow all DECLARE ALIAS statements. This restriction is not enforced for dynamically executed DECLARE TRANSACTION statements.

DECLARE TRANSACTION Statement

You can include multiple DECLARE TRANSACTION statements in a program by linking multiple, separately compiled modules, each with an associated DECLARE TRANSACTION statement. However, the transaction characteristics that the statements specify will not necessarily apply to their modules.

At execution time, when any module starts a transaction, the characteristics declared by that module apply to all modules until the transaction ends. In other words, the DECLARE TRANSACTION statement only specifies characteristics for implicit transactions started by that module; it does not ensure that those characteristics are current when execution begins. Depending on the execution path of your program, this may make it difficult to control the transaction characteristics that apply to a particular module. For instance, if a module does not have an explicit DECLARE TRANSACTION statement and that module starts a transaction, default transaction characteristics apply to all modules until the transaction ends.

When it is important to have particular transaction characteristics apply to a given module, you must be careful to end transactions before program control branches to that module. The SET TRANSACTION statement is best suited to this situation.

- When you use the AND ON clause to start a transaction for more than one database, you should make sure that the DECLARE TRANSACTION statement includes an ON clause for every attached database. If you do not, you cannot use or refer to the databases omitted from the DECLARE TRANSACTION statement in any SQL statement, including SHOW and later DECLARE TRANSACTION statements.
- If you use the BATCH UPDATE clause with DECLARE TRANSACTION statement, SQL will return an error at compile time because BATCH UPDATE is not compatible with two phase commit (2PC).

```
$ sql$ declare transaction batch update;  
%SQL-F-NOBATCHUPDATE, BATCH UPDATE is not allowed without setting  
of SQL$DISABLE_CONTEXT logical name
```

After disabling 2PC then this declare transaction will be accepted. However, Oracle Rdb recommends that BATCH UPDATE transaction be used seldom and with care as they can not be recovered and may leave the database in an unusable state.

```
$ define SQL$DISABLE_CONTEXT TRUE  
$ sql$ declare transaction batch update;
```


DECLARE TRANSACTION Statement

- If you use the DECLARE TRANSACTION statement in a stored module with either the RESERVING table clause or the EVALUATING constraint clause, SQL establishes dependencies on the tables or constraints that you specify. See the CREATE MODULE Statement for a list of statements that can or cannot cause stored procedure invalidation.

See the *Oracle Rdb Guide to SQL Programming* for detailed information about stored procedure dependency types and how metadata changes can cause invalidation of stored procedures.

- By default, a transaction that reserves a table for EXCLUSIVE access does not reserve the LIST (segmented string) area for exclusive access. Because the LIST area is usually shared by many tables, SHARED access is assumed by default to permit updates to the other tables.

This means that when you perform an import operation, or an application updates a table reserved for EXCLUSIVE access, you might notice that the snapshot storage area (.snp) grows. This is because of the I/O to the LIST area that is performed by default when SHARED WRITE mode is in use.

However, if you attach to the database using an SQL ATTACH or IMPORT statement and you specify the RESTRICTED ACCESS clause, then all storage areas are accessed in EXCLUSIVE mode. Use this clause to eliminate the snapshot I/O and related overhead if you are performing a lot of I/O to the LIST storage areas (for example, when you are restructuring the database or dropping a large table containing LIST OF BYTE VARYING columns and data).

Examples

Example 1: Illustrating DECLARE and SET TRANSACTION differences

In the following example, the first executable statement following the DECLARE TRANSACTION statement starts a transaction. In contrast, the subsequent SET TRANSACTION statement itself starts a transaction.

```
SQL> DECLARE TRANSACTION READ WRITE NOWAIT;
SQL> --
SQL> -- Notice the "no transaction is in progress" message:
SQL> --
SQL> SHOW TRANSACTION
Transaction information:
    Statement constraint evaluation is off
```

DECLARE TRANSACTION Statement

```
On the default alias
Transaction characteristics:
    Nowait
    Read Write
Transaction information returned by base system:
no transaction is in progress
- session ID number is 80
SQL> --
SQL> -- The first executable statement following the
SQL> -- DECLARE TRANSACTION statement starts the transaction.
SQL> -- In this case, SELECT is the first executable statement.
SQL> --
SQL> SELECT LAST_NAME FROM CURRENT_SALARY;
LAST_NAME
Toliver
Smith
Dietrich
.
.
.
SQL> --
SQL> -- Note the transaction inherits the read/write characteristics
SQL> -- specified in the DECLARE TRANSACTION statement:
SQL> --
SQL> SHOW TRANSACTION;

Transaction information:
    Statement constraint evaluation is off

On the default alias
Transaction characteristics:
    Nowait
    Read Write
Transaction information returned by base system:
a read-write transaction is in progress
- updates have not been performed
- transaction sequence number (TSN) is 416
- snapshot space for TSNs less than 416 can be reclaimed
- session ID number is 80
SQL> --
SQL> ROLLBACK;
SQL> --
SQL> -- Again, notice the "no transaction is in progress" message:
SQL> --
SQL> SHOW TRANSACTION;

Transaction information:
    Statement constraint evaluation is off
```

DECLARE TRANSACTION Statement

```
On the default alias
Transaction characteristics:
    Nowait
    Read Write
Transaction information returned by base system:
no transaction is in progress
- transaction sequence number (TSN) 416 is reserved
- snapshot space for TSNs less than 416 can be reclaimed
- session ID number is 80
SQL> --
SQL> -- Unlike DECLARE TRANSACTION, the SET TRANSACTION statement
SQL> -- immediately starts a transaction:
SQL> --
SQL> SET TRANSACTION READ ONLY WAIT;
SQL> --
SQL> -- Note the transaction characteristics show the
SQL> -- read-only characteristics:
SQL> --
SQL> SHOW TRANSACTION;
Transaction information:
    Statement constraint evaluation is off
```

```
On the default alias
Transaction characteristics:
    Wait
    Read only
Transaction information returned by base system:
a snapshot transaction is in progress
- all transaction sequence numbers (TSNs) less than 416 are visible
- TSN 416 is invisible
- all TSNs greater than or equal to 417 are invisible
- session ID number is 80
```

Example 2: Using a DECLARE TRANSACTION statement in a context file

The following example shows a context file, `test_declares.sql`, that contains declarations for precompiling source file `test.sco`:

```
DECLARE ALIAS FOR FILENAME personnel;
DECLARE TRANSACTION READ WRITE
    RESERVING EMPLOYEES FOR PROTECTED WRITE,
    JOB_HISTORY FOR PROTECTED WRITE,
    DEPARTMENTS FOR SHARED READ,
    JOBS FOR SHARED READ;
```

The section in the *Oracle Rdb Guide to SQL Programming* about program transportability explains when you may need an SQL context file to support a program that includes SQL statements.

DECLARE TRANSACTION Statement

Example 3: Explicitly setting the isolation level in a DECLARE TRANSACTION statement

In this example, you declare the default characteristics for a read/write transaction, which includes changing the default ISOLATION LEVEL SERIALIZABLE to ISOLATION LEVEL REPEATABLE READ.

```
SQL> DECLARE TRANSACTION READ WRITE ISOLATION LEVEL REPEATABLE READ;
```

Example 4: Reserving a Partition

```
SQL> -- Determine the ordinal position of the EMPLOYEES
SQL> -- partitions.
SQL> SELECT RDB$MAP_NAME, RDB$AREA_NAME, RDB$ORDINAL_POSITION
cont> FROM RDB$STORAGE_MAP_AREAS
cont> WHERE RDB$MAP_NAME='EMPLOYEES_MAP';
  RDB$MAP_NAME          RDB$AREA_NAME
  RDB$ORDINAL_POSITION
EMPLOYEES_MAP          EMPIDS_LOW
                        1
EMPLOYEES_MAP          EMPIDS_MID
                        2
EMPLOYEES_MAP          EMPIDS_OVER
                        3

3 rows selected
SQL> --
SQL> -- Reserve EMPIDS_MID and EMPIDS_OVER for
SQL> -- exclusive write.
SQL> --
SQL> DECLARE TRANSACTION
cont> RESERVING EMPLOYEES PARTITION (2,3)
cont> FOR EXCLUSIVE WRITE;
```

DECLARE Variable Statement

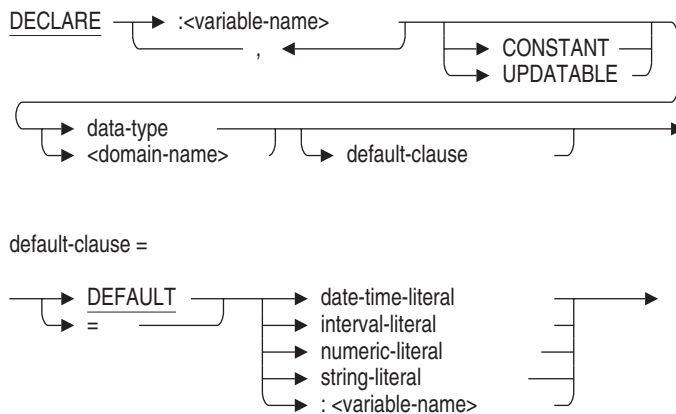
Declares variables that you can use in interactive and dynamic SQL for invoking stored procedures and for testing procedures in modules or embedded SQL programs. For information on declaring variables in compound statements, see the Compound Statement.

Environment

You can use the DECLARE statement:

- In interactive SQL
- In dynamic SQL as a statement to be dynamically executed

Format



Arguments

CONSTANT UPDATABLE

CONSTANT changes the variable into a declared constant that cannot be updated. If you specify CONSTANT, you must also have specified the DEFAULT clause to ensure the variable has a value. CONSTANT also indicates that the variable cannot be used as the target of an assignment or be passed as an expression to a procedure's INOUT or OUT parameter.

DECLARE Variable Statement

UPDATABLE is the default and allows the variable to be modified. An update of a variable can occur due to a SET assignment, an INTO assignment (as part of an INSERT, UPDATE, or SELECT statement), an equality (=) comparison, or as a procedure's OUT or INOUT parameter.

data-type

Specifies the data type assigned to the variable. See Section 2.3 for more information on data types.

default-clause

You can only use references to simple literal values and other declared variables as a default.

domain-name

Specifies the domain name assigned to the variable. The domain supplies the data type and, for interactive SQL, the edit string of the variable.

See Section 2.2.9 for more information on domain names.

variable-name

Specifies the local variable.

Usage Notes

- Variables inside compound statements can be set to NULL. Interactive variables are more like host variables or parameters. You must use indicator variables to set interactive SQL variables to NULL.
- Variables exist until the end of the session or until the UNDECLARE Variable statement is executed. See the UNDECLARE Variable Statement for more information about deleting variable definitions.
- Use the SHOW VARIABLES statement to show the existing variable definitions.
- If the DEFAULT clause is not present, the declared variables initial value is undefined.
- If a list of variables are declared together, the DEFAULT is applied to each variable.
- UPDATABLE is the default for all declared variables.

DECLARE Variable Statement

Example

Example 1: Declaring variables in interactive SQL

```
SQL> DECLARE :X INTEGER;
SQL> DECLARE :Y CHAR(10);
SQL>
SQL> BEGIN
cont>   SET :X = 100;
cont>   SET :Y = 'Active';
cont> END;
SQL> PRINT :X, :Y ;
          X   Y
          100 Active
SQL> SHOW VARIABLES;
X          INTEGER
Y          CHAR(10)
```

Example 2: Using the values of SQLSTATE in an interactive SQL script

The following simple script uses the named `SQLSTATE` variable with the `SIGNAL` statement to make the script easier to read.

```
@SYS$LIBRARY:SQLSTATE
set verify;
begin
signal :SQLSTATE_DATA_ASSIGN ('Error in assignment');
end;
```

When executed the output appears as shown below.

```
SQL> begin
cont> signal :SQLSTATE_DATA_ASSIGN ('Error in assignment');
cont> end;
%RDB-E-SIGNAL_SQLSTATE, routine "(unnamed)" signaled SQLSTATE "22005"
-RDB-I-TEXT, Error in assignment
SQL>
```

DELETE Statement

DELETE Statement

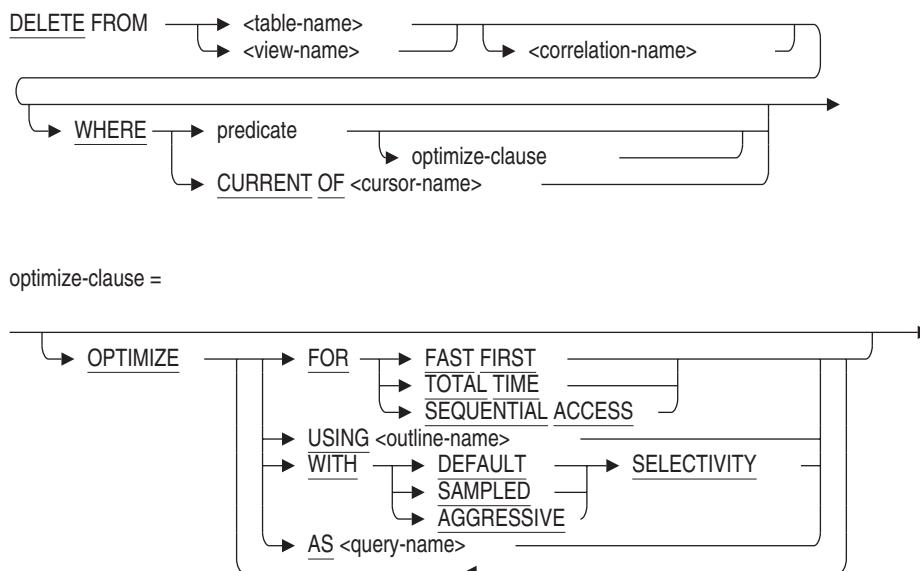
Deletes a row from a table or view.

Environment

You can use the DELETE statement:

- In interactive SQL
- Embedded in host language programs to be precompiled
- As part of a procedure in an SQL module
- In dynamic SQL as a statement to be dynamically executed

Format



Arguments

correlation name

Specifies a name that identifies the table or view in the predicate of the DELETE statement. See Section 2.2.4.1 for more information about correlation names.

DELETE Statement

CURRENT OF cursor-name

If the WHERE clause uses CURRENT OF cursor-name, SQL deletes only the row on which the named cursor is positioned.

The cursor must have been named previously in a DECLARE CURSOR statement, must be open, and must be positioned on a row. In addition, the FROM clause of the SELECT statement within the DECLARE CURSOR statement must refer to the table or view that is the target of the DELETE statement.

OPTIMIZE AS query-name

Assigns a name to the query.

OPTIMIZE FOR

The OPTIMIZE FOR clause specifies the preferred optimizer strategy for statements that specify a select expression. The following options are available:

- **FAST FIRST**

A query optimized for FAST FIRST returns data to the user as quickly as possible, even at the expense of total throughput.

If a query can be cancelled prematurely, you should specify FAST FIRST optimization. A good candidate for FAST FIRST optimization is an interactive application that displays groups of records to the user, where the user has the option of aborting the query after the first few screens. For example, singleton SELECT statements default to FAST FIRST optimization.

If optimization strategy is not explicitly set, FAST FIRST is the default.

- **TOTAL TIME**

If your application runs in batch, accesses all the records in the query, and performs updates or writes a report, you should specify TOTAL TIME optimization. Most queries benefit from TOTAL TIME optimization.

- **SEQUENTIAL ACCESS**

Forces the use of sequential access. This is particularly valuable for tables that use the strict partitioning functionality.

OPTIMIZE USING outline-name

Names the query outline to be used with the DELETE statement even if the outline ID for the query and for the outline are different.

A **query outline** is an overall plan for how a query can be implemented. See the CREATE OUTLINE Statement for additional information.

DELETE Statement

OPTIMIZE WITH

Selects one of three optimization controls: **DEFAULT** (as used by previous versions of Oracle Rdb), **AGGRESSIVE** (assumes smaller numbers of rows will be selected), and **SAMPLED** (which uses literals in the query to perform preliminary estimation on indices).

predicate

If the **WHERE** clause includes a predicate, all the rows of the target table for which the predicate is true are deleted. See Section 2.7 for more information on predicates.

table-name

view-name

Specifies the name of the target table or view from which you want to delete a row.

WHERE

Specifies the rows of the target table or view that will be deleted. If you omit the **WHERE** clause, SQL deletes all rows of the target table or view. You can specify either a predicate or a cursor name in the **WHERE** clause.

Usage Notes

- When specifying a column name, if the column name is the same as a parameter, you should use a correlation name or table name with the column name to avoid confusion with the parameter name.
- The **CURRENT OF** clause in an embedded **DELETE** statement cannot name a cursor based on a dynamic **SELECT** statement. To refer to a cursor based on a dynamic **SELECT** statement in the **CURRENT OF** clause, prepare and dynamically execute the **DELETE** statement as well.
- The **CURRENT OF** clause in an embedded **DELETE** statement cannot name a read-only cursor. See the Usage Notes in the **DECLARE CURSOR Statement** for information about which cursors are read-only.
- You cannot specify the **OPTIMIZE USING** or the **OPTIMIZE AS** clause with the **WHERE CURRENT OF** clause.
- You cannot specify an outline name in a compound-use-statement. See the **Compound Statement** for more information about compound statements.

DELETE Statement

- If an outline exists, Oracle Rdb will use the outline specified in the `OPTIMIZE USING` clause unless one or more of the directives in the outline cannot be followed. SQL issues an error message if the existing outline cannot be used.

If you specify the name of an outline that does not exist, Oracle Rdb compiles the query, ignores the outline name, and searches for an existing outline with the same outline ID as the query. If an outline with the same outline ID is found, Oracle Rdb attempts to execute the query using the directives in that outline. If an outline with the same outline ID is not found, the optimizer selects a strategy for the query for execution.

See the *Oracle Rdb7 Guide to Database Performance and Tuning* for more information regarding query outlines.

Examples

Example 1: Deleting all information about an employee

To delete all the information about an employee, you need to delete rows from several tables within a single transaction. This program fragment deletes the rows from all the result tables that contain information about an employee. Note that all the DELETE operations are included in one transaction so that no employee's records are only partially deleted.

```
        DISPLAY "Enter the ID number of employee".
        DISPLAY "whose records you want to delete:  "
            WITH NO ADVANCING.
        ACCEPT EMP-ID.

EXEC SQL
    DECLARE TRANSACTION READ WRITE
    RESERVING EMPLOYEES      FOR PROTECTED WRITE,
            JOB_HISTORY      FOR PROTECTED WRITE,
            SALARY_HISTORY   FOR PROTECTED WRITE,
            DEGREES          FOR PROTECTED WRITE
END-EXEC

EXEC SQL
    DELETE FROM EMPLOYEES E
    WHERE E.EMPLOYEE_ID = :EMP-ID
END-EXEC

IF SQLCODE < 0 THEN
    EXEC SQL          ROLLBACK          END-EXEC
    GO TO ERROR-PAR
END-IF
```

DELETE Statement

```
EXEC SQL
    DELETE FROM JOB_HISTORY JH
    WHERE JH.EMPLOYEE_ID = :EMP-ID
END-EXEC

IF SQLCODE < 0 THEN
    EXEC SQL          ROLLBACK          END-EXEC
    GO TO ERROR-PAR
END-IF

EXEC SQL
    DELETE FROM SALARY_HISTORY SH
    WHERE SH.EMPLOYEE_ID = :EMP-ID
END-EXEC

IF SQLCODE < 0 THEN
    EXEC SQL          ROLLBACK          END-EXEC
    GO TO ERROR-PAR
END-IF

EXEC SQL
    DELETE FROM DEGREES D
    WHERE D.EMPLOYEE_ID = :EMP-ID
END-EXEC

IF SQLCODE < 0 THEN
    EXEC SQL          ROLLBACK          END-EXEC
    GO TO ERROR-PAR
END-IF
```

Example 2: Deleting selected rows from a table

The following statement deletes all rows from the **EMPLOYEES** table where the employee **SALARY_AMOUNT** is greater than \$75,000. The **EMPLOYEES** and **SALARY_HISTORY** tables are both in the database with the alias **PERS**.

```
SQL> ATTACH 'ALIAS PERS FILENAME personnel';
SQL> DELETE FROM PERS.EMPLOYEES E
cont> WHERE EXISTS ( SELECT *
cont>                   FROM   PERS.SALARY_HISTORY S
cont>                   WHERE  S.EMPLOYEE_ID = E.EMPLOYEE_ID
cont>                   AND    S.SALARY_AMOUNT > 75000
cont>                   ) ;
7 rows deleted
```

Example 3: Deleting rows from a table specifying an outline name

The following example shows the syntax used to define the **DEL_EMP_75000** outline:

DELETE Statement

```
SQL> CREATE OUTLINE DEL_EMP_75000
cont> FROM
cont>   (DELETE FROM EMPLOYEES E
cont>     WHERE EXISTS ( SELECT *
cont>                     FROM SALARY_HISTORY S
cont>                     WHERE S.EMPLOYEE_ID = E.EMPLOYEE_ID
cont>                     AND   S.SALARY_AMOUNT > 75000
cont>                   );
```

The following query specifies the DEL_EMP_75000 outline:

```
SQL> DELETE FROM EMPLOYEES E
cont> WHERE EXISTS ( SELECT *
cont>                 FROM SALARY_HISTORY S
cont>                 WHERE S.EMPLOYEE_ID = E.EMPLOYEE_ID
cont>                 AND   S.SALARY_AMOUNT > 75000
cont>               )
cont> OPTIMIZE USING DEL_EMP_75000;
~S: Outline DEL_EMP_75000 used
.
.
.
7 rows deleted
```

DESCRIBE Statement

DESCRIBE Statement

Writes information about a prepared statement to the SQL Descriptor Area (SQLDA).

The DESCRIBE statement is a dynamic SQL statement. **Dynamic SQL** lets programs accept or generate SQL statements at run time, in contrast to SQL statements that are part of the source code for precompiled programs or SQL module language procedures. Unlike precompiled SQL or SQL module language statements, such dynamically executed SQL statements are not part of a program's source code but are generated while the program runs. Dynamic SQL is useful when you cannot predict the type of SQL statement your program needs to process.

The **SQLDA** is a collection of host language variables used only in dynamic SQL programs. To use the SQLDA, host languages must support pointer variables that provide indirect access to storage by storing the address of data instead of directly storing data in the host language variable. The languages supported by the SQL precompiler that also support pointer variables are Ada, C, and PL/I. Any other language that supports pointer variables can use the SQLDA, but must call SQL module procedures that contain SQL statements instead of embedding the SQL statements directly in source code. The SQLDA provides information about dynamic SQL statements to the program and information about memory allocated by the program to SQL.

The DESCRIBE statement is how SQL writes information that the program uses to the SQLDA. Specifically, the DESCRIBE statement stores in the SQLDA the number and data types of any select list items or parameter markers in a prepared statement.

Appendix D describes in more detail the specific fields of the SQLDA and how programs use it to communicate about select list items and parameter markers in prepared statements.

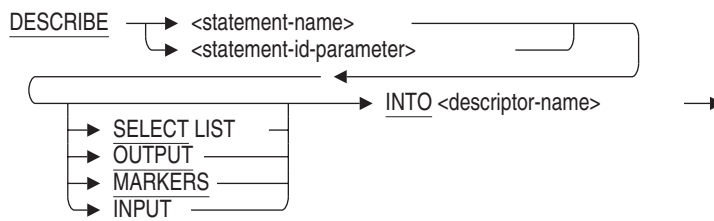
Environment

You can use the DESCRIBE statement:

- Embedded in host language programs to be precompiled
- As part of a procedure in an SQL module

DESCRIBE Statement

Format



Arguments

INTO descriptor-name

Specifies the name of a structure declared in the host language program as an SQLDA to which SQL writes information about select list items, or input or output parameter markers.

Precompiled programs can use the embedded SQL statement INCLUDE SQLDA to automatically insert a declaration of an SQLDA structure, called SQLDA, in the program when it precompiles the program. Programs that use the SQL module language must explicitly declare an SQLDA. Either precompiled or SQL module language programs can explicitly declare additional SQLDAs but must declare them with unique names. For sample declarations of SQLDA structures, see Appendix D.3.

MARKERS

INPUT

Specifies that the DESCRIBE statement writes information about input parameter markers to the SQLDA. The MARKERS or INPUT clause specifies that the DESCRIBE statement writes information about the number and data types of any input parameter markers in the prepared statement to the SQLDA.

Input parameter markers in a prepared statement serve the same purpose as host language variables in nondynamic, embedded SQL statements. The program can use that information in the SQLDA to allocate storage. The program must supply values in that allocated storage. SQL substitutes these values for the parameter markers when it dynamically executes the prepared statement.

DESCRIBE Statement

SELECT LIST OUTPUT

Specifies that the DESCRIBE statement writes information about returned values in a prepared statement to the SQLDA. If you use this clause, the DESCRIBE statement writes information about the number and data types of any returned values in the prepared statement to the SQLDA. The program uses that information to allocate storage for the returned values. The storage allocated by the program then receives the returned values.

The following statements or clauses return values to the DESCRIBE statement:

- Select list items in a SELECT statement
- The following statements within multistatement procedures:
 - Singleton SELECT statement
 - INSERT . . . RETURNING and UPDATE . . . RETURNING statements
 - SET assignment statement
- CALL statement (invoking a stored procedure)
- Dynamic singleton SELECT statement

The default is SELECT LIST (or OUTPUT).

statement-name statement-id-parameter

Specifies the name of a prepared statement. If the PREPARE statement for the dynamically executed statement specifies a parameter, use the same parameter in the DESCRIBE statement instead of an explicit statement name.

You can supply either a parameter or a compile-time statement name. Specifying a parameter lets SQL supply identifiers to programs at run time. Use an integer parameter to contain the statement identifier returned by SQL or a character string parameter to contain the name of the statement that you pass to SQL. See the PREPARE Statement and the DECLARE CURSOR Statement, Dynamic for more details.

Usage Notes

- Programs can set values for any fields in the SQLDA in addition to or instead of having SQL set the values in a DESCRIBE statement. SQL uses the values set by the program.

DESCRIBE Statement

Examples

Example 1: Using the DESCRIBE . . . OUTPUT statement with a prepared SELECT statement

This PL/I program illustrates using the DESCRIBE . . . OUTPUT statement to write information to the SQLDA about the select list items of a prepared SELECT statement. There are no parameter markers in this particular prepared SELECT statement.

After issuing the DESCRIBE statement, the program stores in the SQLDA the addresses of host language variables that will receive values from columns of the result table during FETCH statements.

To shorten the example, this PL/I program is simplified:

- The program includes the SELECT statement to be dynamically executed as part of the source code directly in the PREPARE statement. A program with such coded SQL statements does not need to use dynamic SQL at all, but can simply embed the SELECT statement in a DECLARE CURSOR statement. (A program that must process SQL statements generated as it executes is the only type that requires dynamic SQL.)
- The program declares host language variables for select list items without checking the SQLDA for a description of those items. Typically, an application needs to look in the SQLDA to determine the number and data type of select list items generated by a prepared SELECT statement before allocating storage.
- The program does not use the DESCRIBE . . . INPUT statement to determine if there are any parameter markers in this dynamically executed SELECT statement. In this example, because the SELECT statement is coded in the program, it is clear that there is no need for a DESCRIBE . . . INPUT statement. However, if the SELECT statement is generated at run time, the program may have to determine there if are parameter markers by issuing a DESCRIBE . . . INPUT statement and looking at the value of the SQLD field in the SQLDA.

DESCRIBE Statement

```
CURSOR_EX : PROCEDURE OPTIONS (MAIN);
/*
 * Illustrate the DESCRIBE...SELECT LIST statement using a
 * dynamic SELECT statement:
 *
 * Use a dynamic SELECT statement as the basis for
 * a cursor that displays a join of the EMPLOYEES
 * and SALARY_HISTORY tables on the screen.
 */
declare sys$putmsg external entry
(any, any value, any value, any value);

/* Declare SQL Communications Area: */
EXEC SQL INCLUDE SQLCA;

/* Declare SQL Descriptor Area: */
EXEC SQL INCLUDE SQLDA;

/* Declare the alias: */
EXEC SQL DECLARE ALIAS FILENAME 'SQL$DATABASE';

/*
 * Branch to ERR_HANDLER if the SQLCODE field
 * of the SQLCA is greater than 0:
 */
EXEC SQL WHENEVER SQLERROR GOTO ERR_HANDLER;

/*
 * Declare a cursor named EMP that uses the
 * prepared statement DYN_SELECT:
 */
EXEC SQL DECLARE EMP CURSOR FOR DYN_SELECT;

/* Declare a host structure to receive
 * the results of FETCH statements:
 */
DCL 1 P_REC,
      2 EMPLOYEE_ID   CHAR(5),
      2 FIRST_NAME    CHAR(10),
      2 LAST_NAME     CHAR(14),
      2 SALARY_AMOUNT FIXED BINARY(31);

/* Allocate memory for the SQLDA and
 * set the value of its SQLN field:
 */
SQLSIZE = 10;
ALLOCATE SQLDA SET (SQLDAPTR);
SQLN = 10;
```

DESCRIBE Statement

```
/* Prepare the SELECT statement
 * for dynamic execution directly
 * from a statement string:
 */
EXEC SQL PREPARE DYN_SELECT FROM
      'SELECT E.EMPLOYEE_ID,
            E.FIRST_NAME,
            E.LAST_NAME,
            S.SALARY_AMOUNT
      FROM EMPLOYEES E, SALARY_HISTORY S
      WHERE E.EMPLOYEE_ID = S.EMPLOYEE_ID AND
            S.SALARY_END IS NULL';

/* Write information about the
 * columns of the result table
 * of DYN_SELECT into the SQLDA:
 */
EXEC SQL DESCRIBE DYN_SELECT OUTPUT INTO SQLDA;

/*
 * Assign the addresses of the host language
 * variables that will receive the values of the
 * fetched row to the SQLDATA field
 * of the SQLDA:
 */
SQLDATA(1) = ADDR( EMPLOYEE_ID );
SQLDATA(2) = ADDR( FIRST_NAME);
SQLDATA(3) = ADDR( LAST_NAME );
SQLDATA(4) = ADDR( SALARY_AMOUNT);

/* Open the cursor: */
EXEC SQL OPEN EMP;

/* Fetch the first row of the result table.
 * SQL uses the addresses in the SQLDA
 * to store values from the table into
 * the host language variables.
 */
EXEC SQL FETCH EMP USING DESCRIPTOR SQLDA;

PUT EDIT ('Current Salaries of Employees: ') (SKIP, A, SKIP(2));

/* While the SQLCODE field of the
 * SQLCA is not 100 (NOT_FOUND error):
 */
DO WHILE (SQLCA.SQLCODE = 0);

      /* Display the values from the host language variables: */
      PUT SKIP EDIT
        (EMPLOYEE_ID, ' ', FIRST_NAME, ' ', LAST_NAME, ' ',
         SALARY_AMOUNT)
        (A, A, A, A, A, A, A, F(9));
```

DESCRIBE Statement

```
        /* Fetch another row of the result table: */
        EXEC SQL FETCH EMP USING DESCRIPTOR SQLDA;
END;

/* Close the cursor: */
EXEC SQL CLOSE EMP;

RETURN;

ERR_HANDLER:
    PUT EDIT
    ('Unexpected error, SQLCODE is: ', SQLCA.SQLCODE) (skip, a, f(9));
    CALL SYSPUTMSG(RDB$MESSAGE_VECTOR, 0, 0, 0);
    EXEC SQL WHENEVER SQLERROR CONTINUE;
    EXEC SQL ROLLBACK;
    RETURN;
END CURSOR_EX;
```

See also Example 2 in DECLARE CURSOR Statement, Dynamic.

DISCONNECT Statement

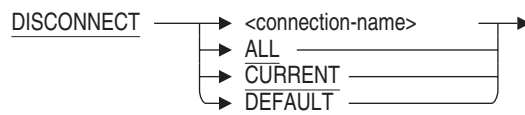
Detaches from declared databases and releases the aliases that you specified in the declarations. The DISCONNECT statement also ends the specified transactions and undoes all the changes you made since those transactions began.

Environment

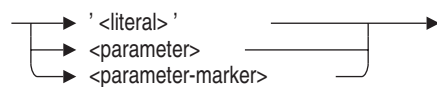
You can use the DISCONNECT statement:

- In interactive SQL
- Embedded in host language programs to be precompiled
- As part of a procedure in an SQL module
- In dynamic SQL as a statement to be dynamically executed

Format



connection-name =



Arguments

ALL

Specifies all active connections.

connection-name

Specifies a name for the association between the group of databases being attached (the environment) and the database and queries that reference them (the session).

DISCONNECT Statement

You can specify the connection name as the following:

- A string literal enclosed in single quotation marks
- A parameter (in module language)
- A variable (in precompiled SQL)

CURRENT

Specifies the current connection.

DEFAULT

Specifies the default connection.

Usage Notes

- Use the DISCONNECT DEFAULT statement instead of the FINISH statement. The FINISH statement is deprecated syntax. Because the DISCONNECT DEFAULT statement performs an automatic rollback, be sure to commit any changes that you want to keep before you execute the DISCONNECT statement.

See the *Oracle Rdb Guide to SQL Programming* for disconnect information with module language procedures.

- You can use SQL connections and explicit calls to DECdtm services to control when you attach and detach from specific databases. By explicitly calling DECdtm system services and associating each database with an SQL connection, you can detach from one database while remaining attached to other databases. For more information, see the *Oracle Rdb7 Guide to Distributed Transactions*.

Examples

Example 1: Using the DISCONNECT statement in interactive SQL

This example in interactive SQL illustrates that the DISCONNECT statement lets you attach a database with the same alias as a previously attached database (in this example the alias is the default). Use the SHOW DATABASE statement to see the database settings.

DISCONNECT Statement

```
SQL> ATTACH 'FILENAME mypers';
SQL> --
SQL> ATTACH 'FILENAME mypers';
This alias has already been declared.
Would you like to override this declaration (No)? no
%SQL-F-DEFDBDEC, A database has already been declared with the default alias
SQL> DISCONNECT DEFAULT;
SQL> ATTACH 'FILENAME mypers';
```

Example 2: Using the DISCONNECT statement in precompiled SQL

This example is taken from the sample program `sql_connections.sc`. To use connections in a program, you must specify the `SQLOPTIONS=(CONNECT)` qualifier on the precompiler command line. This example shows an EXEC SQL DISCONNECT statement that specifies the string literal 'al' for the connection-name and EXEC SQL DISCONNECT statements that specify the keywords ALL and DEFAULT.

```
#include <stdio.h>
#include <string.h>
#include <descrip.h>

char   employee_id1[6];
char   last_name1[16];
char   employee_id2[16];
char   degree[14];
char   employee_id3[16];
char   supervisor[6];
char   employee_id4[6];
char   last_name4[15];

void sys$putmsg();

EXEC SQL INCLUDE SQLCA;
EXEC SQL declare      alias filename personnel;
EXEC SQL declare alias_1 alias filename personnel;
EXEC SQL declare alias_2 alias filename personnel;
EXEC SQL declare alias_3 alias filename personnel;
main()
{
    printf("\n\n***** Disconnect from default *****\n");
    EXEC SQL disconnect default;
    if (SQLCA.SQLCODE != 0) SYS$PUTMSG(&RDB$MESSAGE_VECTOR,0,0,0);
    printf("\n\n");

    printf("***** Establish CONNECTION 1 *****\n");

    EXEC SQL connect to 'alias alias_1 filename personnel' as 'al';
    if (SQLCA.SQLCODE != 0) SYS$PUTMSG(&RDB$MESSAGE_VECTOR,0,0,0);

    printf("***** Insert a record *****\n");
```

DISCONNECT Statement

```
EXEC SQL insert into alias_1.employees (employee_id, last_name)
values ('00301', 'FELDMAN');
  if (SQLCA.SQLCODE != 0) SYS$PUTMSG(&RDB$MESSAGE_VECTOR,0,0,0);

printf("***** Retrieve the record *****\n");
EXEC SQL select employee_id, last_name into :employee_id1,
:last_name1 from alias_1.employees where employee_id = '00301';
  if (SQLCA.SQLCODE != 0) SYS$PUTMSG(&RDB$MESSAGE_VECTOR,0,0,0);
printf("\n\n");
printf ("Employee_id = %s\n",employee_id1);
printf ("Last_name   = %s\n",last_name1);
printf("\n\n");

printf("***** Establish CONNECTION 2 *****\n");
EXEC SQL connect to 'alias alias_2 filename personnel' as 'a2';
  if (SQLCA.SQLCODE != 0) SYS$PUTMSG(&RDB$MESSAGE_VECTOR,0,0,0);

printf("***** Insert a record *****\n");
EXEC SQL insert into alias_2.degrees (employee_id, degree_field)
values ('00901', 'MASTERS');
  if (SQLCA.SQLCODE != 0) SYS$PUTMSG(&RDB$MESSAGE_VECTOR,0,0,0);

printf("***** Retrieve the record *****\n");
EXEC SQL select employee_id, degree_field into :employee_id2,
:degree from alias_2.degrees where employee_id = '00901';
  if (SQLCA.SQLCODE != 0) SYS$PUTMSG(&RDB$MESSAGE_VECTOR,0,0,0);
printf("\n\n");
printf("Employee-id   = %s\n",employee_id2);
printf("Degree       = %s\n",degree);
printf("\n\n");

printf("***** Establish CONNECTION 3 *****\n");
EXEC SQL connect to 'alias alias_3 filename personnel' as 'a3';
  if (SQLCA.SQLCODE != 0) SYS$PUTMSG(&RDB$MESSAGE_VECTOR,0,0,0);

printf("***** Insert a record *****\n");
EXEC SQL insert into alias_3.job_history (employee_id, supervisor_id)
values ('01501', 'Brown');
  if (SQLCA.SQLCODE != 0) SYS$PUTMSG(&RDB$MESSAGE_VECTOR,0,0,0);

printf("***** Retrieve the record *****\n");
EXEC SQL select employee_id, supervisor_id into :employee_id3,
:supervisor from alias_3.job_history where employee_id = '01501';
  if (SQLCA.SQLCODE != 0) SYS$PUTMSG(&RDB$MESSAGE_VECTOR,0,0,0);
printf("\n\n");
printf("Employee-id   = %s\n",employee_id3);
printf("Supervisor   = %s\n ",supervisor);
printf("\n\n");

printf("***** Establish CONNECTION DEFAULT *****\n");
EXEC SQL set connect default ;
  if (SQLCA.SQLCODE != 0) SYS$PUTMSG(&RDB$MESSAGE_VECTOR,0,0,0);
```


DISCONNECT Statement

```
printf("***** Retrieve record with id 00164 *****\n");
EXEC SQL select employee_id, last_name into :employee_id4,
:last_name4 from employees where employee_id = '00164';
if (SQLCA.SQLCODE != 0) SYS$PUTMSG(&RDB$MESSAGE_VECTOR,0,0,0);
printf("\n\n");
printf("Employee_id = %s\n",employee_id4);
printf("Last_name = %s\n",last_name4);
printf("\n\n");

printf("**** DISCONNECT, RECONNECT & TRY TO FIND RECORD ****\n");
strcpy(employee_id1,"");
strcpy(last_name1,"");
EXEC SQL disconnect 'a1';
if (SQLCA.SQLCODE != 0) SYS$PUTMSG(&RDB$MESSAGE_VECTOR,0,0,0);
EXEC SQL connect to 'alias alias_1 filename personnel' as 'a1';
if (SQLCA.SQLCODE != 0) SYS$PUTMSG(&RDB$MESSAGE_VECTOR,0,0,0);
printf("***** Retrieve the record *****\n");
EXEC SQL select employee_id, last_name into :employee_id1,
:last_name1 from alias_1.employees where employee_id = '00301';
if (SQLCA.SQLCODE != 0) SYS$PUTMSG(&RDB$MESSAGE_VECTOR,0,0,0);

printf("***** SHOULD DISPLAY NO RECORD *****\n");
printf("\n\n");
printf("employee_id = %s\n",employee_id1);
printf("last_name = %s\n",last_name1);
printf("\n\n");
printf("***** DISCONNECT ALL CONNECTIONS *****\n");
EXEC SQL disconnect all;
if (SQLCA.SQLCODE != 0) SYS$PUTMSG(&RDB$MESSAGE_VECTOR,0,0,0);
EXEC SQL rollback;
}
```

DROP Statements

DROP Statements

Deletes the database object.

Usage Notes

The following notes apply to all DROP statements except DROP DATABASE.

- You cannot execute the DROP statement when any of the LIST, DEFAULT or RDB\$SYSTEM storage areas are set to read-only. You must first set these storage areas to read/write. Note that in some databases RDB\$SYSTEM will also be the default and list storage area.
- For multischema databases the IF EXISTS clause may not operate as expected because the object is internally deleted using the STORED NAME, which may be different from that specified by the DROP statement. Currently, the IF EXISTS clause assumes that the multischema name and the stored name are identical.
- You must execute the DROP statement in a read/write transaction. If you issue this statement when there is no active transaction, SQL implicitly starts a transaction with characteristics specified in the most recent DECLARE TRANSACTION statement.
- The DROP statement fails when the following are true:
 - The database to which it applies was created with the DICTONARY IS REQUIRED argument.
 - The database was attached using the FILENAME argument.

Under these circumstances, the statement fails with the following error when you issue it:

```
%RDB-E-NO_META_UPDATE, metadata update failed  
-RDMS-F-CDDISREQ, CDD required for metadata updates is not being maintained
```

- An error is reported if the DROP statement is used for an unknown database object. Use the IF EXISTS in SQL command scripts to avoid unwanted error messages.

DROP CATALOG Statement

Deletes the specified catalog definition. You must delete all schemas and definitions contained in a catalog before you can delete that catalog. If other definitions exist that refer to the named catalog, the deletion fails.

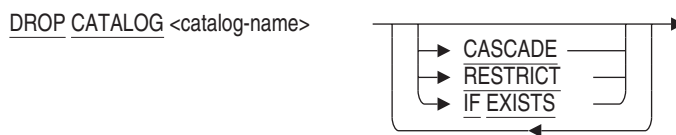
The DROP CATALOG statement lists all schemas and definitions that it is going to delete. You can roll back the statement if you do not want to delete these definitions.

Environment

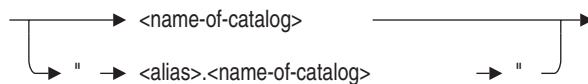
You can use the DROP CATALOG statement:

- In interactive SQL
- Embedded in host language programs to be precompiled
- As part of a procedure in an SQL module
- In dynamic SQL as a statement to be dynamically executed

Format



catalog-name =



Arguments

alias.name-of-catalog

Specifies a name for the attachment to the database. Always qualify the catalog name with an alias if your program or interactive SQL statements refer to more than one database. Separate the name of the catalog from the alias with a period, and enclose the qualified name in double quotation marks.

DROP CATALOG Statement

You must issue a `SET QUOTING RULES` statement before you can qualify a catalog name with an alias.

CASCADE RESTRICT

Performs a restricted delete by default. If you prefer to delete all definitions contained in the catalog, you can specify the `DROP CATALOG CASCADE` statement.

catalog-name

Specifies the module catalog name.

IF EXISTS

Prevents SQL command language from displaying error messages if the referenced object does not exist in the database.

Usage Notes

- You must have `DROP` database privilege in order to drop a catalog from a multischema database.
- You cannot delete a catalog if another user issued a query using that catalog. Users must detach from the database with a `DISCONNECT` statement before you can delete the catalog.
- You cannot delete the catalog `RDB$CATALOG`.

Example

Example 1: Deleting a catalog

The following statement deletes the catalog `DEPT1` associated with the alias `PERSONNEL_ALIAS`:

DROP CATALOG Statement

```
SQL> ATTACH 'ALIAS PERSONNEL_ALIAS FILENAME CORPORATE_DATA';
SQL> SET QUOTING RULES 'SQL99';
SQL> CREATE CATALOG "PERSONNEL_ALIAS.DEPT1";
SQL> SHOW CATALOG;
Catalogs in database PERSONNEL_ALIAS
"PERSONNEL_ALIAS.ADMINISTRATION"
"PERSONNEL_ALIAS.RDB$CATALOG"
"PERSONNEL_ALIAS.DEPT1"
SQL> DROP CATALOG "PERSONNEL_ALIAS.DEPT1";
SQL> SHOW CATALOG;
Catalogs in database PERSONNEL_ALIAS
"PERSONNEL_ALIAS.ADMINISTRATION"
"PERSONNEL_ALIAS.RDB$CATALOG"
SQL> DROP CATALOG "PERSONNEL_ALIAS.RDB$CATALOG";
%SQL-F-NODROPSYSCAT, Catalog "PERSONNEL_ALIAS.RDB$CATALOG" may not be
dropped
SQL>
```

DROP COLLATING SEQUENCE Statement

DROP COLLATING SEQUENCE Statement

Deletes the named collating sequence.

You cannot delete a collating sequence if it is used by the database or by any domain in the database.

Environment

You can use the DROP COLLATING SEQUENCE statement:

- In interactive SQL
- Embedded in host language programs to be precompiled
- As part of a procedure in an SQL module
- In dynamic SQL as a statement to be dynamically executed

Format

DROP COLLATING SEQUENCE → <collation-name> └── IF EXISTS ─┘

Arguments

collation-name

Specifies the collation-name argument you used when creating the collating sequence in the CREATE COLLATING SEQUENCE statement.

IF EXISTS

Prevents SQL command language from displaying error messages if the referenced object does not exist in the database.

Usage Notes

- You must have DROP database privilege in order to drop a collating sequence from a database.
- You may not drop the collating sequence that was made the default for the database.

DROP COLLATING SEQUENCE Statement

```
SQL> create database filename TEST
cont> collating sequence french french;
SQL> drop coll seq french;
%RDB-E-NO_META_UPDATE, metadata update failed
-RDMS-F-COLUSEDDB, the collating sequence named FRENCH is used by the database
```

- You must execute this statement in a read/write transaction. If you issue this statement when there is no active transaction, SQL starts a read/write transaction implicitly.
- Other users are allowed to be attached to the database when you issue the DROP COLLATING SEQUENCE statement.

Examples

Example 1: Creating, then deleting, a French collating sequence

The following example creates a collating sequence using the predefined collating sequence FRENCH. It then uses the SHOW COLLATING SEQUENCE statement to show the defined collating sequence.

The example next deletes the collating sequence using the DROP COLLATING SEQUENCE statement. The SHOW COLLATING SEQUENCE statement shows that the collating sequence no longer exists.

```
SQL> ATTACH 'FILENAME personnel';
SQL> CREATE COLLATING SEQUENCE FRENCH FRENCH;
SQL> --
SQL> SHOW COLLATING SEQUENCE
User collating sequences in database with filename personnel
    FRENCH
SQL> --
SQL> DROP COLLATING SEQUENCE FRENCH;
SQL> --
SQL> SHOW COLLATING SEQUENCE
User collating sequences in database with filename personnel
No collating sequences found
```

Example 2: Deleting a collating sequence used to define a domain or database

The following example shows that you cannot delete a collating sequence if a domain or database is defined using the collating sequence:

```
SQL> CREATE COLLATING SEQUENCE SPANISH SPANISH;
SQL> CREATE DOMAIN LAST_NAME_SPANISH CHAR (14)
cont> COLLATING SEQUENCE IS SPANISH;
SQL> --
SQL> SHOW DOMAIN LAST_NAME_SPANISH
LAST_NAME_SPANISH          CHAR(14)
    Collating sequence: SPANISH
SQL> --
```

DROP COLLATING SEQUENCE Statement

```
SQL> SHOW COLLATING SEQUENCE
User collating sequences in database with filename personnel
SPANISH
SQL> --
SQL> -- You cannot delete the collating sequence because the
SQL> -- domain LAST_NAME_SPANISH, defined using SPANISH, still exists:
SQL> --
SQL> DROP COLLATING SEQUENCE SPANISH;
%RDB-E-NO_META_UPDATE, metadata update failed
-RDMS-F-COLUSEDFLD, the collating sequence named SPANISH is used in
field LAST_NAME_SPANISH
SQL> --
SQL> -- Delete the domain:
SQL> --
SQL> DROP DOMAIN LAST_NAME_SPANISH;
SQL> --
SQL> -- Now you can delete the collating sequence:
SQL> --
SQL> DROP COLLATING SEQUENCE SPANISH;
SQL> --
SQL> SHOW COLLATING SEQUENCE
User collating sequences in database with filename personnel
No collating sequences found
```

DROP CONSTRAINT Statement

Deletes the named constraints.

Environment

You can use the DROP CONSTRAINT statement:

- In interactive SQL
- Embedded in host language programs to be precompiled
- As part of a procedure in an SQL module
- In dynamic SQL as a statement to be dynamically executed

Format

DROP CONSTRAINT → <constraint-name> IF EXISTS

Arguments

constraint-name

Specifies the name of the constraint that you want to delete.

IF EXISTS

Prevents SQL command language from displaying error messages if the referenced object does not exist in the database.

Usage Notes

- You must have DROP table privilege for each table referenced by the table or column constraint. For instances, a FOREIGN KEY constraint will require DROP privilege for the source table as well as the referenced table.
- If the constraint is a column or table constraint, this DROP statement will implicitly execute an ALTER TABLE . . . DROP CONSTRAINT. Refer to ALTER TABLE Statement for more information.

DROP CONSTRAINT Statement

- Attempts to delete a constraint fail if that constraint is in a table involved in a query at the same time. Users must detach from the database with a DISCONNECT statement before you can delete the constraint. When Oracle Rdb first accesses an object such as the constraint, a lock is placed on that object and not released until the user exits the database. If you attempt to update this object, you get a *lock conflict on client* message due to the other users' access to the object.

Example

Example 1: Deleting a constraint

The following statement deletes the SEX_NOT_NULL constraint.

```
SQL> DROP CONSTRAINT SEX_NOT_NULL;
```

DROP DATABASE Statement

Deletes a database.

When this statement executes in Oracle Rdb, SQL deletes all the database root and storage area files associated with the database.

If you specify a repository path name in the DROP DATABASE statement or specify an alias for a database attached with the PATHNAME argument, SQL also deletes the repository directory that contains the database definitions.

Caution

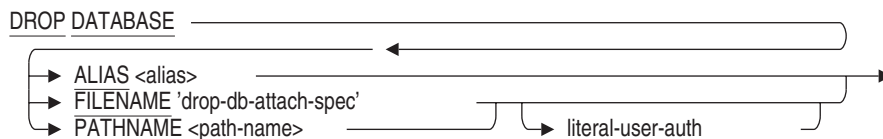
Use the DROP DATABASE statement with care. You cannot use the ROLLBACK statement to cancel a DROP DATABASE statement. When you use this statement, SQL deletes the database root and storage area files, which include all data and all definitions.

Environment

You can use the DROP DATABASE statement:

- In interactive SQL
- Embedded in host language programs to be precompiled
- As part of a procedure in an SQL module
- In dynamic SQL as a statement to be dynamically executed

Format

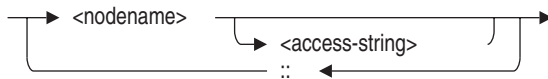


drop-db-attach-spec =

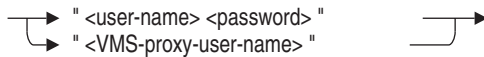


DROP DATABASE Statement

node-spec =



access-string =



literal-user-auth =



Arguments

ALIAS alias

Specifies the alias for an attached database. The DROP DATABASE statement deletes the database and all database root and storage area files associated with the alias.

If the database was declared with the PATHNAME argument, the DROP DATABASE statement also deletes the repository directory that contains the database definitions.

FILENAME 'attach-spec'

Specifies a quoted string containing full or partial information needed to access a database. An attach specification contains the file specification of the .rdb file.

The DROP DATABASE statement deletes the database and all database system files associated with the database root file specification. If you use a partial file specification, SQL uses the standard defaults. The DROP DATABASE statement deletes only the database files, whether or not there is also a repository directory containing database definitions.

literal-user-auth

Specifies the user name and password for access to databases, particularly remote database.

DROP DATABASE Statement

This literal lets you explicitly provide user name and password information in the DROP DATABASE statement.

PATHNAME path-name

Specifies a full or relative repository path name for the repository directory where the database definitions are stored. Use a path name instead of a file specification to delete the repository database definitions from the repository along with the database root and storage area files. See also the DROP PATHNAME Statement.

USER 'username'

Defines a character string literal that specifies the operating system user name that the database system uses for privilege checking.

USING 'password'

Defines a character string literal that specifies the user's password for the user name specified in the USER clause.

Usage Notes

- You must have DROP database privilege to drop a database.
- You cannot delete an Oracle Rdb database when other users are currently attached or is opened using the RMU/OPEN command.
- After Image Journal (.ajj) files are not deleted.

Examples

Example 1: Deleting files only

The following statement deletes the database system files for the database associated with the database personnel.rdb. If this database also had definitions stored in a repository directory, this DROP DATABASE statement would not delete those definitions.

```
SQL> DROP DATABASE FILENAME personnel;
```

Example 2: Deleting files and repository definitions

To delete database files and repository definitions, you must specify a repository path name in the DROP DATABASE statement. This statement deletes the repository directory CDD\$TOP.ACCOUNTING.PERSONNEL in addition to all database root and storage area files associated with it.

```
SQL> DROP DATABASE PATHNAME CDD$TOP.ACCOUNTING.PERSONNEL;
```

DROP DOMAIN Statement

DROP DOMAIN Statement

Deletes a domain definition. If you attached to the database using the `PATHNAME` qualifier, SQL also deletes the domain definition from the repository.

Environment

You can use the `DROP DOMAIN` statement:

- In interactive SQL
- Embedded in host language programs to be precompiled
- As part of a procedure in an SQL module
- In dynamic SQL as a statement to be dynamically executed

Format

`DROP DOMAIN` → `<domain-name>` ┌──────────┐
└── IF EXISTS ─┘ →

Arguments

domain-name

Specifies the name of the domain you want to delete.

IF EXISTS

Prevents SQL command language from displaying error messages if the referenced object does not exist in the database.

Usage Notes

- You must have `DROP` database privilege in order to drop a domain from a database.
- You can delete any named domain. However, you cannot delete a domain that is referred to in a column definition in a table. If you want to delete a domain that is referred to in a column definition, you must first use the `ALTER TABLE` statement to alter or delete the column definition. If the column definition is used in a constraint or index definition, you must first

DROP DOMAIN Statement

delete the constraint or index definition, then alter or delete the column definition.

- Oracle Rdb creates dependencies between stored routines and metadata (like domains) on which they are compiled and stored, therefore, you cannot drop a domain with a routine or trigger dependency. Refer to the CREATE MODULE Statement and CREATE TRIGGER Statement for a list of statements that can or cannot cause stored routine and trigger invalidation.

Refer to the *Oracle Rdb Guide to SQL Programming* for detailed information about stored routine dependency types and how metadata changes can cause invalidation of stored routines.

- If a domain is deleted as part of a DROP SCHEMA CASCADE statement, the domain properties are inherited by any columns defined using the domain.

Examples

Example 1: Deleting a domain not referred to by columns

```
SQL> --
SQL> -- The following CREATE DOMAIN statement creates a domain
SQL> -- that is not used by any columns:
SQL> --
SQL> CREATE DOMAIN ABCD IS CHAR(4);
SQL> --
SQL> -- The SHOW DOMAINS statement shows domain ABCD at the
SQL> -- top of the list:
SQL> --
SQL> SHOW DOMAINS

User domains in database with filename personnel
ABCD                CHAR(4)
ADDRESS_DATA_1_DOM  CHAR(25)
ADDRESS_DATA_2_DOM  CHAR(20)
.
.
.

SQL> --
SQL> -- Now delete the domain:
SQL> --
SQL> DROP DOMAIN ABCD;
SQL> --
SQL> -- The SHOW DOMAINS statement shows that the
SQL> -- domain ABCD has been deleted:
SQL> --
SQL> SHOW DOMAINS
```

DROP DOMAIN Statement

```
User domains in database with filename personnel
ADDRESS_DATA_1_DOM          CHAR(25)
ADDRESS_DATA_2_DOM          CHAR(20)
.
.
.
```

Example 2: Deleting a domain referred to by columns

The following example deletes a domain definition. Because a column refers to the domain definition and a constraint refers to the column, you must first alter the table before deleting the domain.

```
SQL> --
SQL> -- Attempt to delete the domain SEX_DOM. Error messages
SQL> -- indicate that the table EMPLOYEES uses the domain
SQL> -- SEX_DOM, so SEX_DOM cannot yet be deleted:
SQL> --
SQL> DROP DOMAIN SEX_DOM;
%RDB-E-NO_META_UPDATE, metadata update failed
-RDMS-F-RELEXI, field SEX_DOM is used in relation EMPLOYEES
-RDMS-F-FLDNOTDEL, field SEX_DOM has not been deleted
SQL> --
SQL> -- Looking at the EMPLOYEES table shows that SEX is the
SQL> -- column that depends on the domain SEX_DOM. Try
SQL> -- to delete the column SEX; error messages indicate that the
SQL> -- constraint EMP_SEX_VALUES depends on the column SEX:
SQL> --
SQL> ALTER TABLE EMPLOYEES DROP COLUMN SEX;
%RDB-E-NO_META_UPDATE, metadata update failed
-RDMS-F-FLDINCON, field SEX is referenced in constraint EMP_SEX_VALUES
-RDMS-F-RELFNOD, field SEX has not been deleted
from relation EMPLOYEES
SQL> --
SQL> -- Delete the constraint EMP_SEX_VALUES:
SQL> --
SQL> ALTER TABLE EMPLOYEES DROP CONSTRAINT EMP_SEX_VALUES;
SQL> --
SQL> -- Because EMP_SEX_VALUES was the only constraint or index
SQL> -- that depended on the column SEX, you can now delete
SQL> -- the column SEX:
SQL> --
SQL> ALTER TABLE EMPLOYEES DROP COLUMN SEX;
SQL> --
SQL> -- The column SEX in the table EMPLOYEES was the only column in
SQL> -- the database that depended on the domain SEX_DOM, so you can
SQL> -- now delete the domain SEX_DOM:
SQL> --
SQL> DROP DOMAIN SEX_DOM;
SQL>
```

DROP INDEX Statement

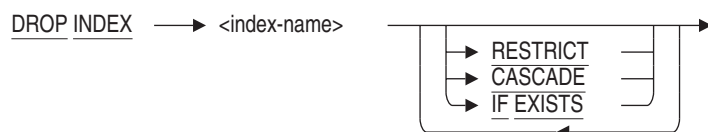
Deletes the specified index definition. If you attach to the database using the `PATHNAME` qualifier, SQL also deletes the index definition from the repository.

Environment

You can use the `DROP INDEX` statement:

- In interactive SQL
- Embedded in host language programs to be precompiled
- As part of a procedure in an SQL module
- In dynamic SQL as a statement to be dynamically executed

Format



Arguments

CASCADE

Specifies that you want SQL to modify any storage map that uses this index to be a `NO PLACEMENT VIA INDEX` storage map.

IF EXISTS

Prevents SQL command language from displaying error messages if the referenced object does not exist in the database.

index-name

Specifies the name of the index definition you want to delete.

RESTRICT

Prevents the removal of an index if it is referenced by any other object within an Oracle Rdb database. `RESTRICT` is the default.

DROP INDEX Statement

Usage Notes

- You must have DROP table privilege in order to drop an index from a table.
- Attempts to delete an index fail if that index is involved in a query at the same time. Users must detach from the database with a DISCONNECT statement before you can delete the index. When Oracle Rdb first accesses an object such as the index, a lock is placed on that object and not released until the user exits the database. If you attempt to update this object, you get a LOCK CONFLICT ON CLIENT message due to the other users' optimized access to the object.

Similarly, while you are deleting an index, users cannot execute queries involving that index until you complete the transaction with a COMMIT or ROLLBACK statement for the DROP statement. The user receives a LOCK CONFLICT ON CLIENT error message.

- CASCADE will implicitly alter any storage map that references this index and change it to a NO PLACEMENT VIA INDEX storage map.
- Any query outline that references the index being dropped will be marked invalid for both RESTRICT and CASCADE options of the DROP INDEX command.
- In a multischema database, the DROP INDEX ... CASCADE statement will be used implicitly to support the DROP SCHEMA ... CASCADE statement. In previous versions of Oracle Rdb this statement would fail if a storage map referenced an index that was to be dropped.

Examples

Example 1: Deleting an index from the default database

```
SQL> ATTACH 'FILENAME personnel';
SQL> DROP INDEX DEG_COLLEGE_CODE;
SQL> COMMIT;
```

Example 2: Deleting an index from one of several attached databases

```
SQL> ATTACH 'FILENAME personnel';
SQL> ATTACH 'ALIAS MF FILENAME mf_personnel';
SQL> ATTACH 'ALIAS CORP FILENAME corporate_data';
SQL> SET QUOTING RULES 'SQL99';
SQL> DROP INDEX "CORP.ADMINISTRATION".PERSONNEL.EMP_EMPLOYEE_ID;
SQL> COMMIT;
```

DROP MODULE Statement

Deletes a module from an Oracle Rdb database.

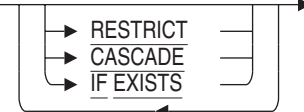
Environment

You can use the DROP MODULE statement:

- In interactive SQL
- Embedded in host language programs to be precompiled
- As part of a procedure in an SQL module
- In dynamic SQL as a statement to be dynamically executed

Format

`DROP MODULE` → <module-name>



Arguments

CASCADE

Specifies that you want SQL to invalidate all objects that refer to routines in the module and then delete that module definition. This is known as a cascading delete. If you delete a module referenced by a stored routine with a routine or language-semantic dependency, SQL also marks the affected stored routine as invalid.

IF EXISTS

Prevents SQL command language from displaying error messages if the referenced object does not exist in the database.

module-name

Identifies the name of the module.

RESTRICT

Prevents the removal of a stored routine definition when the function or procedure is referenced by any other object within an Oracle Rdb database. RESTRICT is the default.

DROP MODULE Statement

Usage Notes

- To execute this statement, you need the DROP privilege on the module you want to delete.
- Attempts to delete a module will fail if the objects in a procedure or function of a stored module are involved in a query at the same time. Users must detach from the database with a DISCONNECT statement before you can delete the module. When SQL first accesses an object such as a module, a lock is placed on that object and not released until the users exit the database.

If you attempt to update this object, you get a *lock conflict on client* message due to the other users' access to the object.

Similarly, while you are deleting a module, users cannot execute queries involving the procedure or function of a module until you complete the transaction with a COMMIT or ROLLBACK statement for the DROP statement. The user receives a LOCK CONFLICT ON CLIENT error message.

- If a table has a computed-by column whose definition invokes a stored function, and if that stored function is being deleted, the column is set to COMPUTE NULL.
- If a module is deleted, invalidating a stored routine, and then the module is redefined, use of the invalid routine attempts to revalidate the routine references. Use the ALTER MODULE statement to revalidate these modules.

Examples

Example 1: Removing a module from an Oracle Rdb database

```
SQL> DROP MODULE employee_salary;
```

Example 2: Observing the DROP MODULE ... CASCASE action

This example demonstrates that dependencies may exist between the module being dropped and other database objects such as routines and triggers. The script uses SET FLAGS with the WARN_INVALID option so that the database administrator is informed of any affected objects. In this case a rollback is used to undo the DROP MODULE ... CASCADE as the affects might damage the application environment.

DROP MODULE Statement

```
SQL> start transaction read write;
SQL>
SQL> create module FIRST_MODULE
cont>     function GET_TIME ()
cont>     returns TIME (2);
cont>     return CURRENT_TIME (2);
cont> end module;
SQL>
SQL> create module SECOND_MODULE
cont>     procedure PRINT_TRACE (in :arg varchar(40));
cont>     begin
cont>         trace GET_TIME(), ': ', :arg;
cont>     end;
cont> end module;
SQL>
SQL> create table SAMPLE_TABLE
cont>     (ident integer,
cont>     descr char(100));
SQL> create trigger SAMPLE_TABLE_TRIGGER
cont>     after insert on SAMPLE_TABLE
cont>     (trace GET_TIME(), ': ', SAMPLE_TABLE.descr)
cont>     for each row;
SQL>
SQL> commit;
SQL>
SQL> set flags 'warn_invalid';
SQL> drop module FIRST_MODULE restrict;
%RDB-E-NO_META_UPDATE, metadata update failed
-RDMS-E-OBJ_INUSE, object "GET_TIME" is referenced by SAMPLE_TABLE_TRIGGER. (usage: Trigger)
-RDMS-E-MODNOTDEL, module "FIRST_MODULE" has not been deleted
SQL> drop module FIRST_MODULE cascade;
~Xw: Trigger "SAMPLE_TABLE_TRIGGER" marked invalid
~Xw: Routine "PRINT_TRACE" marked invalid
SQL>
SQL> rollback;
```

DROP OUTLINE Statement

DROP OUTLINE Statement

Deletes a query outline.

Environment

You can use the DROP OUTLINE statement:

- In interactive SQL
- Embedded in host language programs to be precompiled
- As part of a procedure in an SQL module
- In dynamic SQL as a statement to be dynamically executed

The DROP OUTLINE statement allows the user to specify that an existing outline should be removed from the database.

Format

DROP OUTLINE → <outline-name> IF EXISTS

Arguments

IF EXISTS

Prevents SQL command language from displaying error messages if the referenced object does not exist in the database.

outline-name

Specifies the name of the outline you want to delete.

Usage Notes

- To delete an outline, you must have the DROP privilege for every table referenced by the outline.
- The DROP OUTLINE statement is an online operation (other users can be attached to the database when an outline is deleted). However, a query outline cannot be deleted when the outline is being referenced in another transaction. If you issue a DROP OUTLINE statement while another

DROP OUTLINE Statement

transaction is referencing the outline, the transaction finishes and then the outline is deleted.

Examples

Example 1. Deleting an outline

```
SQL> DROP OUTLINE MY_OUTLINE;
```

DROP PATHNAME Statement

DROP PATHNAME Statement

Deletes the repository definitions. It does not delete the physical database files.

Environment

You can use the DROP PATHNAME statement:

- In interactive SQL
- Embedded in host language programs to be precompiled
- As part of a procedure in an SQL module
- In dynamic SQL as a statement to be dynamically executed

Format

DROP PATHNAME → <path-name> →

Arguments

path-name

Specifies the repository path name for the schema definitions.

Specify either a full path name or a relative path name. If you use a relative path name, the current default repository directory must be defined to include all the path name segments that precede the relative path name.

Examples

Example 1: Deleting a path name with the DROP PATHNAME statement

The following example deletes CDD\$TOP.SQL.DEPT3, a repository directory, and all its descendants. It does not delete the database system files or data that corresponds to that path name.

```
SQL> DROP PATHNAME "CDD$TOP.SQL.DEPT3";
```

DROP PROFILE Statement

Drops a profile definition.


Environment

You can use the DROP PROFILE statement:

- In interactive SQL
- Embedded in host language programs
- As part of a procedure in an SQL module or other compound statement
- In dynamic SQL as a statement to be dynamically executed

Format

DROP PROFILE → <profilename>



Arguments

CASCADE

This option causes all user definitions to be altered to remove the reference to this profile.

IF EXISTS

Prevents SQL command language from displaying error messages if the referenced object does not exist in the database.

RESTRICT

If the profile is used by a user in the database, the DROP PROFILE statement will fail. This is the default.

DROP PROFILE Statement

Usage Notes

- You must have SECURITY database privilege in order to drop a profile from a database.
- Profile names are, by default, in uppercase. If they were defined in mixed case or with other special characters, use the SET DIALECT 'SQL99' or SET QUOTING RULES 'SQL99' statement to enable delimited identifiers. Then, use quotation marks (" ") around the name in the DROP PROFILE statement.

Examples

Example 1: Using Delimited Identification Mixed-Case Profile Names

```
SQL> DROP PROFILE Decision_Support;
%RDB-E-NO_META_UPDATE, metadata update failed
-RDMS-E-PRFNEXISTS, a quota does not exist with the name "DECISION_SUPPORT"
SQL> SET DIALECT 'SQL99';
SQL> DROP PROFILE "Decision_Support";
SQL> COMMIT;
```

Example 2: Using CASCADE to remove assigned profiles from users

This example demonstrates that there may be dependencies between profiles and user objects. The CASCADE action will remove the profile from all users to which is assigned.

DROP PROFILE Statement

```
SQL> create profile DECISION_SUPPORT
cont>     comment is 'restrictions for read-only users'
cont>     default transaction read only
cont>     transaction modes (read only, shared);
SQL>
SQL> show profile DECISION_SUPPORT;
      DECISION_SUPPORT
Comment:      restrictions for read-only users
              Transaction modes (read only, shared)
              Default transaction read only
SQL>
SQL> create user FREEMAN
cont>     identified externally
cont>     profile DECISION_SUPPORT;
SQL>
SQL> show user FREEMAN;
      FREEMAN
      Identified externally
      Account is unlocked
      Profile: DECISION_SUPPORT
      No roles have been granted to this user
SQL>
SQL> drop profile DECISION_SUPPORT restrict;
%RDB-E-NO_META_UPDATE, metadata update failed
-RDMS-E-PRFINUSE, entry "DECISION_SUPPORT" is referenced by user "FREEMAN"
SQL>
SQL> drop profile DECISION_SUPPORT cascade;
SQL>
SQL> show user FREEMAN;
      FREEMAN
      Identified externally
      Account is unlocked
      No roles have been granted to this user
SQL>
SQL> commit;
```

DROP ROLE Statement

DROP ROLE Statement

Drops a role previously created with the CREATE ROLE or GRANT statement.


Environment

You can use the DROP ROLE statement:

- In interactive SQL
- Embedded in host language programs
- As part of a procedure in an SQL module or other compound statement
- In dynamic SQL as a statement to be dynamically executed

Format

`DROP ROLE` → `<role-name>`



Arguments

CASCADE

Drops the specified role from the database and deletes all references to this role that exist in other roles and access control lists (ACLs).

IF EXISTS

Prevents SQL command language from displaying error messages if the referenced object does not exist in the database.

RESTRICT

Drops the specified role. If there are any references to this role in another role or ACL, then the DROP ROLE statement fails.

The RESTRICT clause is the default.

role-name

An existing role-name in the database (such as one created with the CREATE ROLE statement). You cannot specify one of the predefined roles. See the Usage Notes for details.

DROP ROLE Statement

Usage Notes

- You must have the SECURITY privilege on the database to drop a role.
- The special roles BATCH, DIALUP, INTERACTIVE, LOCAL, NETWORK, and REMOTE are granted by the OpenVMS operating system when the user process is created. Therefore, these roles are reserved names and cannot be used as the role-name in the DROP ROLE statement.

Examples

Example 1: Dropping a Role from the Database

```
SQL> SHOW ROLES;
Roles in database with filename mf_personnel.rdb
  DOCUMENTATION
SQL> DROP ROLE DOCUMENTATION RESTRICT;
SQL> SHOW ROLES;
Roles in database with filename mf_personnel.rdb
No Roles Found
```

Drop Routine Statement

Drop Routine Statement

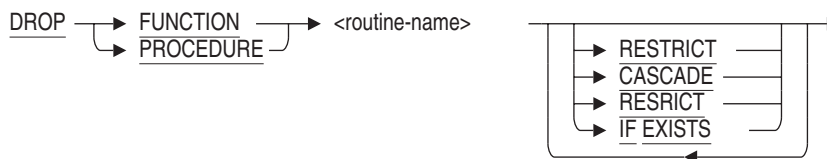
Deletes a routine definition, external or stored, from an Oracle Rdb database. **External routine** refers to both external functions and external procedures. **Stored routine** refers to both stored functions and stored procedures.

Environment

You can use the DROP FUNCTION and DROP PROCEDURE statements:

- In interactive SQL
- Embedded in host language programs to be precompiled
- As part of a procedure in an SQL module
- In dynamic SQL as a statement to be dynamically executed

Format



Arguments

CASCADE

Deletes the routine definition even when there are dependencies on the specified routine. Any referencing routines are marked invalid.

IF EXISTS

Prevents SQL command language from displaying error messages if the referenced object does not exist in the database.

DROP FUNCTION routine-name

Identifies the name of the external or stored function definition in the Oracle Rdb database.

DROP PROCEDURE routine-name

Identifies the name of the external or stored procedure definition in the Oracle Rdb database.

Drop Routine Statement

RESTRICT

Prevents the removal of an external or stored routine definition when the routine is referenced by any other object within an Oracle Rdb database.

RESTRICT is the default.

Usage Notes

- You must have DROP privilege on the routine in order to drop a function or procedure from a database. If the routine was created using CREATE MODULE then you must have DROP privilege on the owning module in order to drop the routine.
- SQL does not store the external routine's executable image in an Oracle Rdb database. Instead, it stores information that points to the external routine, such as its name and location, so that SQL can automatically invoke it from within a query execution.
- Before you can delete a routine in a module, you must have ALTER privileges on the module containing the routine that you want to delete.
- Computed-by columns are set to COMPUTE NULL in tables that reference a function that has been deleted by a DROP FUNCTION CASCADE statement.

You can alter the table and delete the computed-by column. At some future point, you can then alter the table and create a new computed-by column using the same name but with a different computed-by expression.

Examples

Example 1: Deleting an external function definition from an Oracle Rdb database

If you want to alter an external function definition, you must first delete it and then create it again with the changes you plan. This example shows how to delete the COSINE_F function.

```
SQL> DROP FUNCTION cosine_f RESTRICT;
```

Drop Routine Statement

Example 2: Deleting a routine from a stored module

The `DROP FUNCTION` and `DROP PROCEDURE` statements can be used to drop routines from a stored module. If the routine is referenced by other objects then the `CASCADE` option may be required to successfully drop the routine.

See also the `DROP FUNCTION` and `DROP PROCEDURE` clauses of `ALTER MODULE` which can be used to perform the same task.

This example removes a function from the stored module `TIME_ROUTINES` that is no longer in use.

```
SQL> set dialect 'sql99';
SQL> create database filename junk;
SQL>
SQL> create module TIME_ROUTINES
cont>
cont>     function GET_TIME ()
cont>     returns TIME (2);
cont>     return CURRENT_TIME (2);
cont>
cont>     function DAY_OF_WEEK (in :dt date)
cont>     returns VARCHAR(10);
cont>     return case EXTRACT (weekday from :dt)
cont>         when 1 then 'Monday'
cont>         when 2 then 'Tuesday'
cont>         when 3 then 'Wednesday'
cont>         when 4 then 'Thursday'
cont>         when 5 then 'Friday'
cont>         when 6 then 'Saturday'
cont>         when 7 then 'Sunday'
cont>         else '***'
cont>     end;
cont>
cont> end module;
SQL>
SQL> show module TIME_ROUTINES;
Information for module TIME_ROUTINES

Header:
TIME_ROUTINES
No description found
Module ID is: 1

Routines in module TIME_ROUTINES:
    DAY_OF_WEEK
    GET_TIME

SQL> drop function GET_TIME cascade;
SQL> show module TIME_ROUTINES;
Information for module TIME_ROUTINES
```


Drop Routine Statement

```
Header:  
TIME_ROUTINES  
No description found  
Module ID is: 1  
  
Routines in module TIME_ROUTINES:  
    DAY_OF_WEEK  
  
SQL>
```

DROP SCHEMA Statement

DROP SCHEMA Statement

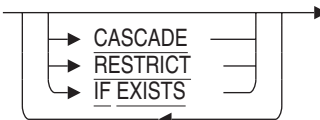
Deletes a schema and all the definitions that it contains.

Environment

You can use the DROP SCHEMA statement:

- In interactive SQL
- Embedded in host language programs to be precompiled
- As part of a procedure in an SQL module
- In dynamic SQL as a statement to be dynamically executed

Format

`DROP SCHEMA <schema-name>` 

The diagram shows a box containing three options: CASCADE, RESTRICT, and IF EXISTS. Each option has a right-pointing arrow to its left and a left-pointing arrow to its right, indicating they are optional keywords that can be included in the statement.

Arguments

CASCADE

Deletes all other definitions (views, constraints, tables, sequences, indexes, and triggers) that refer to the named schema and then deletes that schema definition. This is known as a cascading delete.

If you specify the CASCADE keyword, SQL deletes all definitions contained by the schema before deleting the schema.

If you do not specify the CASCADE keyword, the schema must be empty.

IF EXISTS

Prevents SQL command language from displaying error messages if the referenced object does not exist in the database.

RESTRICT

Returns an error message if other definitions refer to the named schema. The DROP SCHEMA RESTRICT statement will not delete a schema until you have deleted all other definitions that refer to the named schema. The DROP SCHEMA statement specifies an implicit RESTRICT by default.

DROP SCHEMA Statement

schema-name

Specifies the schema name. You must qualify the schema name with catalog and alias names if the schema is not in the default catalog and database. For more information about schema names, see Section 2.2.15.

Usage Notes

- To delete a schema, you must have the same authorization identifier as that schema or your user name must match the schema name.
- You must have DROP database privilege in order to drop a schema from a multischema database.
- If you try to delete a schema without first deleting views, constraints, indexes, and triggers that refer to that schema, SQL issues the following error message:

```
SQL> ATTACH 'ALIAS MS_ALIAS FILENAME MS_TESTDB';
SQL> SET QUOTING RULES 'SQL99';
SQL> SET CATALOG '"MS_ALIAS.MS_TESTCATALOG"';
SQL> DROP SCHEMA "MS_ALIAS.MS_TESTSCHEMA";
%RDB-E-NO_META_UPDATE, metadata update failed
-RDMS-E-SCHEMAINUSE, schema MS_TESTSCHEMA currently in use
```

You can avoid the error message by deleting all the definitions that refer to the named schema before deleting the schema, or by specifying DROP SCHEMA CASCADE.

- You cannot delete the schema RDB\$SCHEMA.

Examples

Example 1: Deleting a schema with implicit RESTRICT

In the following example, the user must delete the definitions that refer to the schema RECRUITING before deleting the schema itself.

After setting the default schema to RECRUITING and the default catalog to ADMINISTRATION, the user can qualify each definition name with only the alias CORP.

DROP SCHEMA Statement

```
SQL> ATTACH 'ALIAS CORP FILENAME CORPORATE_DATA';
SQL> SET CATALOG '"CORP.ADMINISTRATION"';
SQL> SET SCHEMA '"CORP.ADMINISTRATION".RECRUITING';
SQL> SET QUOTING RULES 'SQL92';
SQL> DROP SCHEMA "CORP.RECRUITING";
%RDB-E-NO_META_UPDATE, metadata update failed
-RDMS-E-SCHEMAINUSE, schema RECRUITING currently in use
SQL> DROP TABLE "CORP.CANDIDATES";
SQL> DROP TABLE "CORP.COLLEGES";
%RDB-E-NO_META_UPDATE, metadata update failed
-RDMS-F-CONEXI, relation COLLEGES is referenced in constraint DEGREES_FOREIGN3
-RDMS-F-RELNOTDEL, relation COLLEGES has not been deleted
SQL> DROP TABLE "CORP.DEGREES";
%RDB-E-NO_META_UPDATE, metadata update failed
-RDMS-F-TRGEXI, relation DEGREES is referenced in trigger
EMPLOYEE_ID_CASCADE_DELETE
SQL> DROP TABLE "CORP.RESUMES";
%RDB-E-NO_META_UPDATE, metadata update failed
-RDMS-F-TRGEXI, relation RESUMES is referenced in trigger
EMPLOYEE_ID_CASCADE_DELETE
-RDMS-F-RELNOTDEL, relation RESUMES has not been deleted
SQL> --
SQL> -- The trigger is part of another schema, PERSONNEL. Since this
SQL> -- is not the default schema, the user qualifies the trigger name
SQL> -- with schema and names.
SQL> --
SQL> DROP TRIGGER "CORP.ADMINISTRATION".PERSONNEL.EMPLOYEE_ID_CASCADE_DELETE;
SQL> DROP CONSTRAINT "CORP.DEGREES_FOREIGN3";
%RDB-E-NO_META_UPDATE, metadata update failed
-RDMS-F-CONDELVIAREL, constraint DEGREES_FOREIGN3 can only be deleted by
changing or deleting relation DEGREES
SQL> DROP TABLE "CORP.DEGREES";
SQL> DROP TABLE "CORP.RESUMES";
SQL> DROP TABLE "CORP.COLLEGES";
SQL> DROP SCHEMA "CORP.RECRUITING";
```

Example 2: Deleting a schema with CASCADE

In the following example, SQL deletes the definitions that refer to the schema ACCOUNTING, then deletes the schema itself:

```
SQL> DROP SCHEMA "CORP.ACCOUNTING" CASCADE;
Domain "CORP.ADMINISTRATION".ACCOUNTING.BUDGET is also being dropped.
Domain "CORP.ADMINISTRATION".ACCOUNTING.CODE is also being dropped.
SQL>
```

DROP SEQUENCE Statement

Drops a specified sequence.

Environment

You can use the DROP SEQUENCE statement:

- In interactive SQL
- Embedded in host language programs
- As part of a procedure in an SQL module or other compound statement
- In dynamic SQL as a statement to be dynamically executed

Format

DROP SEQUENCE <sequence-name>



Arguments

CASCADE

The CASCADE clause specifies that you want SQL to invalidate all objects that refer to the sequence and then delete the sequence definition. If you delete a sequence referenced by a stored routine or trigger with a routine or language-semantic dependency, SQL also marks the affected stored routine or trigger as invalid.

IF EXISTS

Prevents SQL command language from displaying error messages if the referenced object does not exist in the database.

RESTRICT

The RESTRICT clause prevents the removal of a sequence definition (the DROP SEQUENCE statement fails) when the sequence is referenced by any other object within the Oracle Rdb database.

The RESTRICT clause is the default.

DROP SEQUENCE Statement

sequence-name

An existing sequence name in the database. To specify lowercase characters or characters not in the SQL repertoire, enclose the sequence name in single quotation marks (').

Usage Notes

- You must have the DROP database privilege on the sequence to drop a sequence from a database.
- When you drop a sequence, the reserved space in the database root file becomes available for reuse by the next sequence created.
- Because Oracle Rdb creates dependencies between stored sequences and metadata (like tables) on which they depend, you can delete a table with a routine or language-semantic dependency if you specify CASCADE but you cannot with RESTRICT. In the case of DROP TABLE CASCADE, when the table referenced in a stored routine is deleted, the stored routine is marked as invalid. In the case of DROP TABLE RESTRICT, because the statement fails when you attempt to delete a table referenced in a stored routine, the dependent stored routine is not invalidated. See the CREATE MODULE Statement for a list of statements that can or cannot cause stored routine invalidation.

See the *Oracle Rdb Guide to SQL Programming* for detailed information about stored routine dependency types and how metadata changes can cause invalidation of stored routines.

- Oracle Rdb creates dependencies between sequences and other database objects, such as tables and routines, which depend upon those definitions. For example, you can delete a sequence with a dependency if you specify CASCADE but you cannot with RESTRICT. In the case of DROP SEQUENCE . . . CASCADE, when the sequence referenced in a stored routine is deleted, the routine is marked as invalid. In the case of DROP SEQUENCE . . . RESTRICT, the statement fails when the dependency is detected and the dependent routine is not invalidated. See the CREATE MODULE Statement for a list of statements that may cause stored routine invalidation.

DROP SEQUENCE Statement

Examples

Example 1: Dropping a Sequence

```
SQL> SHOW SEQUENCE;
Sequences in database with filename mf_personnel.rdb
  EMPID
SQL> DROP SEQUENCE EMPID;
SQL> SHOW SEQUENCE;
Sequences in database with filename mf_personnel.rdb
  No Sequences Found
SQL>
```

DROP STORAGE MAP Statement

DROP STORAGE MAP Statement

Deletes the specified storage map definition.

Environment

You can use the DROP STORAGE MAP statement:

- In interactive SQL
- Embedded in host language programs to be precompiled
- As part of a procedure in an SQL module
- In dynamic SQL as a statement to be dynamically executed

Format

`DROP STORAGE MAP` → `<map-name>` IF EXISTS →

Arguments

IF EXISTS

Prevents SQL command language from displaying error messages if the referenced object does not exist in the database.

map-name

Specifies the name of the storage map you want to delete.

Usage Notes

- You must have DROP table privilege in order to drop a storage map from a table.
- When the storage map is dropped the table is implicitly mapped to the default storage area.
- You cannot delete a storage map that refers to a table that contains data. If you attempt to do so, you receive an error message.
- The underlying storage map is deleted when you use DROP TABLE.

DROP STORAGE MAP Statement

- Attempts to delete a storage map fail if that storage map refers to a table that is involved in a query at the same time. Users must detach from the database with a DISCONNECT statement before you can delete the storage map. When Oracle Rdb first accesses an object such as a table, a lock is placed on that object and not released until the users exit the database. If you attempt to update this object, you get a LOCK CONFLICT ON CLIENT message due to the other users' access to the object.

Similarly, while you are deleting a storage map, users cannot execute queries involving the table that the storage map refers to until you complete the transaction with a COMMIT or ROLLBACK statement for the DROP statement. The users receive a *lock conflict on client* error message.

- Other users are allowed to be attached to the database when you issue the DROP STORAGE MAP statement.

Examples

Example 1: Deleting a storage map in interactive SQL

This example deletes a storage map. You cannot delete a storage map that refers to a table that contains data.

```
SQL> ATTACH 'FILENAME mf_personnel';
SQL> DROP STORAGE MAP WORK_STATUS_MAP;
%RDB-E-NO_META_UPDATE, metadata update failed
-RDMS-F-RELNOTEMPTY, relation WORK_STATUS has data in it
SQL> DELETE FROM WORK_STATUS;
3 rows deleted
SQL> DROP STORAGE MAP WORK_STATUS_MAP;
SQL>
```

DROP SYNONYM Statement

DROP SYNONYM Statement

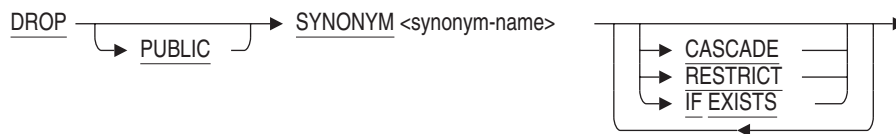
Drops a synonym definition.

Environment

You can use the DROP SYNONYM statement:

- In interactive SQL
- Embedded in host language programs
- As part of a procedure in an SQL module or other compound statement
- In dynamic SQL as a statement to be dynamically executed

Format



Arguments

CASCADE

Specifies that you want SQL to delete the synonym definition even if other database objects reference this name. This might later cause errors when executing queries. Stored functions, stored procedures, and triggers that reference this name will be marked as invalid.

IF EXISTS

Prevents SQL command language from displaying error messages if the referenced object does not exist in the database.

PUBLIC

This optional clause is provided for compatibility with the Oracle database server. It is currently not used by Oracle Rdb. Its presence or absence may be used by future releases. Oracle Corporation recommends you use the **PUBLIC** keyword in applications.

DROP SYNONYM Statement

RESTRICT

Specifies that you want SQL to abort the DROP statement if it detects any database object referencing this name. This is the default.

synonym-name

The name of an existing synonym you want to drop.

Usage Notes

- You must have REFERENCES privilege on the referenced object to drop a synonym for that object. Because domains do not have access control, no other privileges are required to drop synonyms for domains.
- You must have database DROP privilege to execute the DROP SYNONYM statement.

Example

Example 1: Dropping a Synonym

```
SQL> DROP PUBLIC SYNONYM employees CASCADE;
```

DROP TABLE Statement

DROP TABLE Statement

Deletes the specified table definition.

If you use the `PATHNAME` qualifier when you attach to the database, the `DROP TABLE` statement also deletes the table definition from the repository.

Environment

You can use the `DROP TABLE` statement:

- In interactive SQL
- Embedded in host language programs to be precompiled
- As part of a procedure in an SQL module
- In dynamic SQL as a statement to be dynamically executed

Format

`DROP TABLE <table-name>`



Arguments

CASCADE

Specifies that you want SQL to delete all other definitions (constraints, indexes, modules, storage maps, triggers, and views) that refer to the named table and then delete that table definition. This is known as a cascading delete. For stored routines or triggers with a routine or language-semantic dependency, SQL also marks the affected routines and triggers as invalid.

IF EXISTS

Prevents SQL command language from displaying error messages if the referenced object does not exist in the database.

RESTRICT

Specifies that you want SQL to delete only the named table definition. If constraints, modules, triggers, or views are defined that refer to the named table, SQL does not delete the table. If there are indexes or storage maps that

DROP TABLE Statement

refer to the named table, SQL deletes the table and storage map and does not issue an error.

table-name

Specifies the name of the table definition you want to delete.

Usage Notes

- You must have DROP privilege on the table in order to drop that table from a database.
- You cannot delete a table when there are other active transactions involving the table. That is, you must have exclusive access to the table.
- Attempts to delete a table will fail if that table is involved in a query at the same time.
- If you do not specify either the CASCADE keyword or the RESTRICT keyword, SQL executes a restricted delete by default.
- The CASCADE option will invalidate all active queries that reference this table. If using an ORACLE dialect then this is also true for RESTRICT.
- When SQL first accesses an object such as the table, a lock is placed on that object and not released until the user exits the database.

If you are using Oracle Rdb and attempt to update this object, you get a lock conflict on client message due to the other user's access to the object.

Similarly, while you are deleting a table, users cannot execute queries involving that table until you completed the transaction with a COMMIT or ROLLBACK statement for the DROP statement. If you are using Oracle Rdb, users receive a lock conflict on client error message.

- Performance of DROP TABLE statements varies widely, depending on how the storage area file containing the table was defined. In multfile databases, performance is much slower if the base storage area was created with the PAGE FORMAT IS MIXED storage area parameter.

When a table contains one or more LIST OF BYTE VARYING columns, the DROP TABLE statement must read each row in the table and record the pointers for all LIST values. This list is processed at COMMIT time to delete the LIST column data. Therefore, the database administrator must also allow for this time when dropping the table.

DROP TABLE Statement

Reserving the table for EXCLUSIVE WRITE is recommended because the dropped LIST columns will require that each row in the table be updated and set to NULL - it is this action which queues the pointers for commit time processing. This reserving mode will eliminate snapshot file I/O, lower lock resources and reduce virtual memory usage.

As the LIST data is stored outside the table performance may be improved by attaching to the database with the RESTRICTED ACCESS clause, which has the side effect of reserving all the LIST storage areas for EXCLUSIVE access and therefore eliminates snapshot I/O during the delete of the LIST data.

- Other users are allowed to be attached to the database when you issue the DROP TABLE statement.
- If a view definition refers to a table you want to delete, you must delete that view definition before you delete the table, or specify CASCADE.
- If a trigger definition refers to a table you want to delete, you must delete that trigger definition before you delete the table, or specify CASCADE.
- Because Oracle Rdb creates dependencies between stored routines and metadata (like tables) on which they depend, you can delete a table with a routine or language-semantic dependency if you specify CASCADE but you cannot with RESTRICT. In the case of DROP TABLE CASCADE, when the table referenced in a stored routine is deleted, the stored routine is marked as invalid. In the case of DROP TABLE RESTRICT, because the statement fails when you attempt to delete a table referenced in a stored routine, the dependent stored routine is not invalidated. See the CREATE MODULE Statement for a list of statements that can or cannot cause stored routine invalidation.

See the *Oracle Rdb Guide to SQL Programming* for detailed information about stored routine dependency types and how metadata changes can cause invalidation of stored routines.

- The DROP TABLE statement marks any query outline that refers to the deleted table as invalid.
- A computed-by column is altered to COMPUTE NULL if it references a persistent base table, global temporary table, or local temporary table that has been deleted by a DROP TABLE CASCADE statement. For example:

DROP TABLE Statement

```
SQL> CREATE TABLE t1 (col1 INTEGER,
cont>                    col2 INTEGER);
SQL> --
SQL> CREATE TABLE t2 (x INTEGER,
cont>                    y COMPUTED BY (SELECT COUNT(*) FROM
cont>                    t1 WHERE t1.col1 = t2.x));
SQL> --
SQL> -- Assume values have been inserted into the tables.
SQL> --
SQL> SELECT * FROM t1;
      COL1      COL2
      ----      ----
         1         100
         1         101
         1         102
         2         200
         3         300

5 rows selected
SQL> SELECT * FROM t2;
      X      Y
      --      --
         1         3
         3         1

2 rows selected
SQL> --
SQL> DROP TABLE t1 CASCADE;
Computed Column Y in table T2 is being set to NULL.
SQL> SELECT * FROM t2;
      X      Y
      --      --
         1      NULL
         3      NULL
```

You can alter the table and delete the computed-by column. At some future point, you can then alter the table and create a new computed-by column using the same name but with a different computed-by expression.

However, a computed-by column is not set to NULL if it references a declared local temporary table that has been deleted by a **DROP TABLE CASCADE** statement. An exception is raised if you query the declared local temporary table in this situation.

Examples

Example 1: Deleting a table from an attached database

```
SQL> ATTACH 'ALIAS PERS FILENAME personnel';
SQL> DROP TABLE PERS.DEGREES;
SQL> COMMIT;
```

DROP TABLE Statement

Example 2: Deleting a table and definitions that reference it from the default database

```
SQL> ATTACH 'FILENAME corporate_data';
SQL> DROP TABLE ADMINISTRATION.PERSONNEL.EMPLOYEES CASCADE;
View ADMINISTRATION.PERSONNEL.REVIEW_DATE is also being dropped.
View ADMINISTRATION.PERSONNEL.CURRENT_INFO is also being dropped.
View ADMINISTRATION.PERSONNEL.CURRENT_SALARY is also being dropped.
View ADMINISTRATION.PERSONNEL.CURRENT_JOB is also being dropped.
Constraint ADMINISTRATION.RECRUITING.DEGREES_FOREIGN2 is also being dropped.
Constraint ADMINISTRATION.PERSONNEL.EMPLOYEES_PRIMARY_EMPLOYEE_ID is also
being dropped.
Constraint ADMINISTRATION.PERSONNEL.EMP_SEX_VALUES is also being dropped.
Constraint ADMINISTRATION.PERSONNEL.HOURLY_HISTORY_FOREIGN1 is also being
dropped.
Constraint ADMINISTRATION.PERSONNEL.JOB_HISTORY_FOREIGN1 is also being
dropped.
Constraint ADMINISTRATION.RECRUITING.RESUMES_FOREIGN2 is also being dropped.
Constraint ADMINISTRATION.PERSONNEL.SALARY_HISTORY_FOREIGN1 is also being
dropped.
Constraint ADMINISTRATION.PERSONNEL.STATUS_CODE_VALUES is also being dropped.
Index ADMINISTRATION.PERSONNEL.EMP_LAST_NAME is also being dropped.
Index ADMINISTRATION.PERSONNEL.EMP_EMPLOYEE_ID is also being dropped.
Trigger ADMINISTRATION.PERSONNEL.EMPLOYEE_ID_CASCADE_DELETE is also being
dropped.
Trigger ADMINISTRATION.PERSONNEL.STATUS_CODE_CASCADE_UPDATE is also being
dropped.
```

DROP TRIGGER Statement

Deletes a trigger definition from the physical database and, if the database was attached with PATHNAME, from the repository.

Environment

You can use the DROP TRIGGER statement:

- In interactive SQL
- Embedded in host language programs to be precompiled
- As part of a procedure in an SQL module
- In dynamic SQL as a statement to be dynamically executed

Format

DROP TRIGGER → <trigger-name> IF EXISTS

Arguments

IF EXISTS

Prevents SQL command language from displaying error messages if the referenced object does not exist in the database.

trigger-name

Specifies the name of the trigger to be deleted.

Usage Notes

- To delete a trigger, you must have DELETE access to the table for which the trigger is defined.
- You must have DROP table privilege in order to drop a trigger for a table.
- Attempts to delete a trigger fail if that trigger is involved in a query at the same time. Users must detach from the database with a DISCONNECT statement before you can delete the trigger. When Oracle Rdb first accesses an object such as the table accessed by the trigger, a lock is placed on that object and not released until the user exits the database. If you attempt to

DROP TRIGGER Statement

update this object, you get a LOCK CONFLICT ON CLIENT message due to the other users' access to the object.

Similarly, while you are deleting a trigger, users cannot execute queries involving tables referred to by the trigger until you have completed the transaction with a COMMIT or ROLLBACK statement for the DROP statement. The user receives *lock conflict on client* error message.

- Other users are allowed to be attached to the database when you issue the DROP TRIGGER statement.

Examples

Example 1: Deleting the EMPLOYEE_ID_CASCADE_DELETE trigger

```
SQL> ATTACH 'FILENAME personnel';
SQL> SHOW TRIGGERS
User triggers in database with filename PERSONNEL
  COLLEGE_CODE_CASCADE_UPDATE
  EMPLOYEE_ID_CASCADE_DELETE
  STATUS_CODE_CASCADE_UPDATE
SQL> DROP TRIGGER EMPLOYEE_ID_CASCADE_DELETE;
SQL> SHOW TRIGGERS
User triggers in database with filename PERSONNEL
  COLLEGE_CODE_CASCADE_UPDATE
  STATUS_CODE_CASCADE_UPDATE
SQL>
```

DROP USER Statement

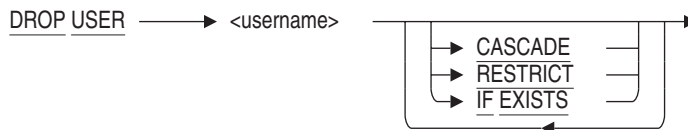
Removes the entry (such as one created with the CREATE USER or GRANT statement) for a user name or special user class from the database.

Environment

You can use the DROP statement:

- In interactive SQL
- Embedded in host language programs
- As part of a procedure in an SQL module or other compound statement
- In dynamic SQL as a statement to be dynamically executed

Format



Arguments

CASCADE

The CASCADE clause drops the specified user from the database and deletes all references to this user that exist in the access control lists (ACLs), modules, and schemas. If the PUBLIC user is dropped, ACLs are not processed to remove the PUBLIC entry.

RESTRICT

The RESTRICT clause drops the specified user. If there are any references to this user in another ACL, then the DROP USER statement fails.

The RESTRICT clause is the default.

username

An existing user name in the database.

DROP USER Statement

Usage Notes

- You must have the `SECURITY` privilege on the database to drop a user.
- You can display existing users defined for a database by issuing a `SHOW SYSTEM USERS` or `SHOW USERS` statement.

Example

Example 1: Dropping a User

```
SQL> SHOW USER
Users in database with filename mf_personnel.rdb
  JSMITH
  NSTUART
SQL> DROP USER JSMITH;
SQL> SHOW USER
Users in database with filename mf_personnel.rdb
  NSTUART
SQL>
```

DROP VIEW Statement

Deletes the specified view definition. When the DROP VIEW statement executes, SQL deletes the view definition from the database. If you attach to the database using the PATHNAME qualifier, SQL also deletes the view definition from the repository.

Environment

You can use the DROP VIEW statement:

- In interactive SQL
- Embedded in host language programs to be precompiled
- As part of a procedure in an SQL module
- In dynamic SQL as a statement to be dynamically executed

Format

`DROP VIEW <view-name>` 

The diagram shows a box with three options: CASCADE, RESTRICT, and IF EXISTS. Arrows point from the text 'CASCADE', 'RESTRICT', and 'IF EXISTS' to the box. A larger arrow points from the box to the right, indicating the flow of the statement.

Arguments

CASCADE

Specifies that you want SQL to delete all other view definitions that refer to the named view and then delete that view definition. This is known as a cascading delete. If you delete a view referenced by a stored routine or trigger with a routine or language-semantic dependency, SQL also marks the affected routines and triggers as invalid.

IF EXISTS

Prevents SQL command language from displaying error messages if the referenced object does not exist in the database.

RESTRICT

Specifies that you want SQL to delete only the named view definition. If there are other views, triggers, or routines that refer to the named view, the deletion fails. RESTRICT is the default.

DROP VIEW Statement

view-name

Specifies the name of the view definition you want to delete.

Usage Notes

- You must have DROP privilege on the view in order to drop that view from a database.
- Because Oracle Rdb creates dependencies between stored routines and metadata (like views) on which they depend, you can delete a view with a routine or language-semantic dependency if you specify CASCADE but you cannot with RESTRICT. In the case of DROP VIEW CASCADE, when the view referenced in a stored routine is deleted, the stored routine is marked as invalid. In the case of DROP VIEW RESTRICT, because the statement fails when you attempt to delete the view referenced in a stored routine, the dependent stored routine is not invalidated. Refer to the CREATE MODULE Statement for a list of statements that can or cannot cause stored routine invalidation.

Refer to the *Oracle Rdb Guide to SQL Programming* for detailed information about stored routine dependency types and how metadata changes can cause invalidation of stored routines.

- If a deleted view is referenced in a computed-by column, the computed-by column is altered to COMPUTE NULL.

Examples

Example 1: Deleting a view definition

The following example deletes the view definition CURRENT_INFO:

```
SQL> DROP VIEW CURRENT_INFO;  
SQL> COMMIT;
```

Example 2: Deleting a view with dependent views

This example shows that SQL will not automatically delete any views that refer to the view named in the DROP VIEW statement. You must use the CASCADE keyword to delete a view with dependent views.

DROP VIEW Statement

```
SQL> DROP VIEW CURRENT_JOB;
%RDB-E-NO_META_UPDATE, metadata update failed
-RDMS-F-VIEWINVIEW, view CURRENT_JOB is referenced by view CURRENT_INFO
-RDMS-F-VIEWNOTDEL, view CURRENT_JOB has not been deleted

SQL> DROP VIEW CURRENT_JOB CASCADE;
View CURRENT_INFO is also being dropped.
SQL> COMMIT;
```

Example 3: Adding new definitions to a database

When updating metadata definitions using a predefined SQL script it sometimes required to remove objects that may not be present in all databases being maintained. Adding a DROP VIEW, for instance, will result in an error as shown here.

```
SQL> drop view CURRENT_INFO;
%SQL-F-RELNOTDEF, Table CURRENT_INFO is not defined in database or schema
SQL> create view CURRENT_INFO
cont> ...etc...
```

By using the IF EXISTS clause the error message is suppressed and makes for a less confusing execution of the maintenance script.

```
SQL> drop view CURRENT_INFO if exists;
SQL> create view CURRENT_INFO
cont> ...etc...
```

EDIT Statement

EDIT Statement

Calls an editor that lets you modify the SQL statements you issued within a terminal session.

SQL supports a variety editors, some of which are:

- EDT
- DEC Text Processing Utility (DECTPU) editors on OpenVMS, such as EVE
- Language-Sensitive Editor (LSE) on OpenVMS, which is based on DECTPU and provides templates that guide you in entering syntactically correct statements

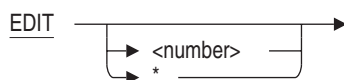
To invoke an editor other than the default, you must define the SQL\$EDIT logical name. See the Usage Notes section for details.

You can use the editor you choose with your usual initialization file to modify your previous SQL statements, construct your next statement or group of statements, or include a file with other statements.

Environment

You can issue the EDIT statement only in interactive SQL.

Format



Arguments

* (asterisk)

Specifies a wildcard character. If you use the * (asterisk) wildcard character, SQL includes in the editing buffer the number of statements specified in the last SET EDIT KEEP statement. If you do not use the SET EDIT KEEP statement, EDIT * puts the last 20 statements in your editing buffer. If you omit the * (asterisk) wildcard character, SQL includes the last statement issued in the editing buffer.

EDIT Statement

number

Specifies the number of previous statements you want to edit, up to the number specified in the last SET EDIT KEEP statement. If you specify zero as the number, then SQL does not include any statements in the editing buffer. If you omit the number argument, SQL includes the last statement issued in the editing buffer.

Usage Notes

- When you use the EDIT statement, the following sequence occurs:
 1. SQL invokes the editor specified by the SQL\$EDIT logical name and initializes the editor according to your initialization file for that editor, if any. If you do not have an initialization file, SQL uses the system default editor.
 2. SQL places the statements you asked for in the editing buffer.
If you are using an editor other than EDT, DECTPU, or LSE, SQL places the statements in a temporary file and spawns a subprocess to execute the command you specified in the SQL\$EDIT logical name.
 3. The SQL prompt (SQL>) disappears and is replaced by the normal display for the editor.
 4. You can now edit the SQL statements.
If you are using the EDT, DECTPU, or LSE editor, SQL automatically executes all the statements in the main editing buffer when you exit from the editor. If you are using an editor other than EDT, DECTPU, or LSE, you are prompted whether or not you want to execute the command lines in the main editing buffer when you exit the editor. A later Usage Note explains how to bypass this prompt and execute the command lines automatically with other editors.
If you quit from the editor, SQL returns to the command level and displays the SQL prompt (SQL>) without executing a statement.
- You do not need to do anything to specify EDT as the editor to use within interactive SQL because it is the OpenVMS system default editor. To use DECTPU, it must be installed on your system, and you must define the logical name SQL\$EDIT. To use LSE, it must be installed on your system, and you must define the logical names SQL\$EDIT and LSE\$ENVIRONMENT.

EDIT Statement

```
$ ! To specify DECTPU as your editor in interactive SQL:
$ DEFINE SQL$EDIT TPU
$ !
$ ! To specify LSE as your editor in interactive SQL:
$ DEFINE SQL$EDIT LSE
$ DEFINE LSE$ENVIRONMENT -
_ $ SYSS$COMMON:[SYSLIB]LSE$SYSTEM_ENVIRONMENT.ENV
```

Then, when you type `EDIT` in an SQL session, SQL calls the editor specified by the `SQL$EDIT` logical name. If `SQL$EDIT` is not defined or is defined to be something other than `TPU` or `LSE`, then SQL invokes the `EDT` editor when you issue the `EDIT` command. If SQL cannot find the `TPU` or `LSE` shareable image, it invokes `EDT`.

- If you specify an editor based on `DECTPU` for use in interactive SQL (through the `SQL$EDIT` logical), you cannot always read or write files from the editing buffer created when you issue the interactive SQL statement `EDIT`.
 - In `EVE` editors, the `INCLUDE` command to read a file into the default editing buffer fails. To work around this problem, you must use the `GET FILE` command to place the file in another buffer and copy the buffer to the `MAIN` buffer that SQL executes upon exiting from the editor.
 - In all editors based on `DECTPU`, the `DECTPU WRITE_FILE` command (`WRITE` in `EVE`) to write the default editing buffer fails. You must copy the default buffer to another buffer and write that buffer to a file.
- If you execute an SQL statement and then execute the `HELP` statement to read the help text, an `EDIT` statement puts only the original SQL statement in the editing buffer, not the `HELP` statement.
- Interactive SQL users can recall the 20 most recent command lines using the up and down arrow keys or the `Ctrl/B` key sequence.
 - The up arrow key recalls lines in sequence from most recent to least recent.
 - The `Ctrl/B` key sequence also recalls lines in sequence from most recent to least recent.
 - After you recalled prior lines, the down arrow key allows you to recall more recently entered lines.
- The `EDIT` statement does not save any operating system invocation statements or executable statements in the buffer of statements to edit.

EDIT Statement

- If you are using an editor other than EDT, DECTPU, or LSE, you are prompted whether or not you want to execute the command lines in the main editing buffer when you exit the editor. You can bypass this prompt by setting the SQL\$EDIT_TWO logical name.

The SQL\$EDIT_TWO logical name can be set to true so that the editor accepts an input file followed by an output file. The editor edits the output file and inserts the contents of the input file. Writing out the output file signals SQL to execute the command lines. In order for the SQL\$EDIT_TWO logical name to be useful, the SQL\$EDIT logical name must also be set.

If the SQL\$EDIT_TWO logical name is not set to true, then the editor is invoked with only one file specification and, upon exiting, you are prompted whether or not you want to execute the command lines in the main editing buffer.

Examples

Example 1: Correcting a misspelled statement

1. Make a mistake:

```
SQL> SELECT JOB_TITLE FROM JOSB;
%SQL-F-RELNOTDEF, Table JOSB is not defined in schema
SQL>
```

2. Invoke the editor:

```
SQL> EDIT
```

3. When in the editor, change JOSB to JOBS. See the manual for the editor you are using for detailed editing instructions.
4. Exit from the editor. SQL automatically executes the contents of the editing buffer.

```
* EXIT
SELECT JOB_TITLE FROM JOBS;
Associate Programmer
Clerk
Assistant Clerk
Department Manager
Dept. Supervisor
.
.
.
```

END DECLARE Statement

END DECLARE Statement

Delimits the end of a host language variable declaration section in a precompiled program.

Environment

You can use the `END DECLARE` statement embedded in host language programs to be precompiled.

Format

```
EXEC SQL → BEGIN DECLARE SECTION → ;  
└──────────────────────────────────────────────────────────────────────────────────┘  
└──────────────────────────────────────────────────────────────────────────────────┘  
└──────────────────────────────────────────────────────────────────────────────────┘  
EXEC SQL → END DECLARE SECTION → ;
```

Arguments

BEGIN DECLARE SECTION

Delimits the beginning of a host language variable declaration.

END DECLARE SECTION

Delimits the end of host language variable declarations.

;(semicolon)

Terminates the `BEGIN DECLARE` and `END DECLARE` statements.

Which terminator you use depends on the language in which you are embedding the host language variable. The following table shows which terminator to use:

Host Language	Required SQL Terminator	
	BEGIN DECLARE Statement	END DECLARE Statement
COBOL	END-EXEC	END-EXEC
FORTRAN	None required	None required
Ada, C, Pascal, or PL/I	;(semicolon)	;(semicolon)

END DECLARE Statement

host language variable declaration

Specifies a variable declaration embedded within a program.

See Section 2.2.13 for more information on host language variable definitions.

Usage Notes

- The ANSI/ISO SQL standard specifies that host language variables used in embedded SQL statements must be declared within a pair of embedded SQL `BEGIN DECLARE . . . END DECLARE` statements. If ANSI/ISO compliance is important for your application, you should include all declarations for host language variables used in embedded SQL statements within a `BEGIN DECLARE . . . END DECLARE` block.
- SQL does not require that you enclose host language variables with `BEGIN DECLARE` and `END DECLARE` statements. SQL does, however, issue a warning message if both of the following conditions exist:
 - Your program includes a section delimited by `BEGIN DECLARE` and `END DECLARE` statements.
 - You refer to a host language variable that is declared outside the `BEGIN DECLARE` and `END DECLARE` section.
- In addition to host language variable declarations, you can include other host language statements in a `BEGIN DECLARE . . . END DECLARE` section. See Section 2.2.13 and the `BEGIN DECLARE` Statement for more details.

Examples

Example 1: Declaring a host language variable within a `BEGIN . . . END DECLARE` block

The following example shows portions of a PL/I program. The first part of the example declares the host language variable `LNAME` within the `BEGIN DECLARE` and `END DECLARE` statements. The semicolon is necessary as a terminator because the language is PL/I.

The second part of the example shows a singleton `SELECT` statement that specifies a one-row result table. The statement assigns the value in the row to the previously declared host language variable `LNAME`.

END DECLARE Statement

```
EXEC SQL
BEGIN DECLARE SECTION;
  DECLARE LNAME char(20);
EXEC SQL
END DECLARE SECTION;
.
.
.
EXEC SQL
SELECT FIRST_NAME
  INTO :LNAME
  FROM EMPLOYEES
  WHERE EMPLOYEE_ID = "00164";
```

Execute (@) Statement

In SQL, the at sign (@) means execute. When you type @ and the name of an indirect command file, SQL executes the statements in that file as if you typed them one-at-a-time at the SQL prompt (SQL>). The command file must be a text file that contains SQL statements.

The default file extension for an indirect command file is .SQL.

You can use the SET VERIFY statement to display the commands in the file as they execute.

SQL recognizes a special SQL command file called SQLINI.SQL, which contains SQL statements to be issued before SQL displays the SQL prompt (SQL>). If this file exists, SQL executes the commands in the file first, before displaying the prompt and accepting your input. If you define the logical name to point to a general initialization file, SQL uses this file. Otherwise, it looks for SQLINI.SQL in the current default directory.

Environment

You can issue the execute (@) statement only in interactive SQL.

Format

@<file-spec>

Arguments

file-spec

Specifies the name of an indirect command file. You can use either a full file specification, a file name, or a logical name on OpenVMS. If you use a file name, SQL looks in the current default directory for a file by that name. The file must contain valid SQL statements.

Execute (@) Statement

Usage Notes

Interactive SQL interprets any command line that begins with an at sign (@) as the start of a command file invocation. This is true even if the at sign is a continuation of a string literal from the previous line, which can lead to confusing results.

```
SQL> INSERT INTO EMPLOYEES (CITY) VALUES ('AtSign -
cont> @City')
%SQL-F-FILEACCERR, Error parsing name of file City')
-RMS-F-SYN, file specification syntax error
SQL> --
SQL> -- You can avoid errors by breaking your statement line elsewhere:
SQL> --
SQL> INSERT INTO EMPLOYEES (CITY) VALUES
cont> ('AtSign - @City');
1 row inserted
```

Examples

Example 1: Storing interactive SQL statements in a startup file

You can use an indirect command file to specify characteristics of your SQL terminal session. This example assumes that SQLINI is defined as a logical name that points to the file setup.sql. The file contains the following SQL statements:

```
SET VERIFY;
SET EDIT KEEP 5; -- This line will be displayed on the terminal
```

SQL executes the file when you invoke interactive SQL.

```
$ SQL$
SQL> SET EDIT KEEP 5; -- This line will be displayed on the terminal
SQL>
```

When it executes, setup.sql turns on the indirect command file display and limits the number of statements saved by SQL for editing to five.

Execute (@) Statement

Example 2: Executing frequently used queries

The file EMPADDR.SQL contains the following SQL statements:

```
-- This command file generates information for a mailing list.
--
ATTACH 'FILENAME personnel';
SET OUTPUT MAILLIST.DOC
SELECT FIRST_NAME, MIDDLE_INITIAL, LAST_NAME,
       ADDRESS_DATA_1, ADDRESS_DATA_2, CITY, STATE, POSTAL_CODE
FROM EMPLOYEES;
--
-- Execute the file by using the following command:
--
@EMPADDR
```

Example 3: Using a logical name to run a command file

If you define COUNT to be a logical name, you can use the command @COUNT to execute the statements in the file, even if the file is located in a directory other than the default directory. The file COUNT.SQL contains the following SQL statements:

```
-- This command file counts the rows in
-- each table of the personnel database.
--
SET NOVERIFY;
SELECT 'Count of Employees -----> ', COUNT (*) FROM EMPLOYEES;
SELECT 'Count of Jobs -----> ', COUNT (*) FROM JOBS;
SELECT 'Count of Degrees -----> ', COUNT (*) FROM DEGREES;
SELECT 'Count of Salary_History --> ', COUNT (*) FROM SALARY_HISTORY;
SELECT 'Count of Job_History -----> ', COUNT (*) FROM JOB_HISTORY;
SELECT 'Count of Work_Status -----> ', COUNT (*) FROM WORK_STATUS;
SELECT 'Count of Departments -----> ', COUNT (*) FROM DEPARTMENTS;
SELECT 'Count of Colleges -----> ', COUNT (*) FROM COLLEGES;
```

The following example shows how to execute the file and the output:

```
$ SQL
SQL> @COUNT;

Count of Employees ----->          100
1 row selected

Count of Jobs ----->                15
1 row selected

Count of Degrees ----->            166
1 row selected

.
.
.
```

EXECUTE Statement

EXECUTE Statement

Dynamically executes a previously prepared statement.

The EXECUTE statement is a dynamic SQL statement. Dynamic SQL lets programs accept or generate SQL statements at run time, in contrast to SQL statements that are part of the source code for precompiled programs or SQL module language procedures. Unlike precompiled SQL or SQL module language statements, such dynamically executed SQL statements are not necessarily part of a program's source code, but can be generated while the program is running. Dynamic SQL is useful when you cannot predict the type of SQL statement your program will need to process.

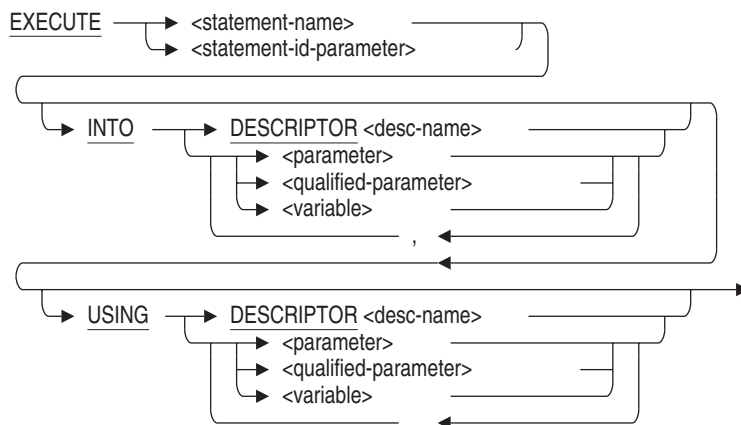
If a program needs to dynamically execute a statement more than once, the statement should be prepared first with the PREPARE statement and executed each time with the EXECUTE statement. SQL does not parse and compile prepared statements every time it dynamically executes them with the EXECUTE statement.

Environment

You can use the EXECUTE statement:

- Embedded in host language programs to be precompiled
- As part of a procedure in an SQL module

Format



EXECUTE Statement

Arguments

INTO DESCRIPTOR descriptor-name

Specifies an SQLDA descriptor that contains addresses and data types that specify output parameters or variables.

The descriptor must be a structure declared in the host language program as an SQLDA. If the program is precompiled and uses the embedded SQL statement `INCLUDE SQLDA`, the name of the structure is simply `SQLDA`. Programs can use multiple SQLDAs, but must explicitly declare them with names other than `SQLDA`.

Programs can always use the `INTO DESCRIPTOR` clause of the `EXECUTE` statement whether or not the statement string contains output parameter markers, as long as the value of the `SQLD` field in the `SQLDA` corresponds to the number of output parameter markers. SQL updates the `SQLD` field with the correct number of output parameter markers when it processes the `DESCRIBE` statement for the statement string.

INTO parameter

INTO qualified-parameter

INTO variable

Specifies output parameters or variables whose values are returned by a successful `EXECUTE` statement.

When you specify a list of parameters or variables, the number of parameters in the list must be the same as the number of output parameter markers in the statement string of the prepared statement. If SQL determines that a statement string had no output parameter markers, the `INTO` clause is not allowed.

statement-name

statement-id-parameter

Specifies the name of a prepared statement. You can supply either a parameter or a compile-time statement name. Specifying a parameter lets SQL supply identifiers to programs at run time. Use an integer parameter to contain the statement identifier returned by SQL or a character string parameter to contain the name of the statement that you pass to SQL.

If the `PREPARE` statement for the dynamically executed statement specifies a parameter, use that same parameter in the `EXECUTE` statement instead of an explicit statement name.

EXECUTE Statement

USING DESCRIPTOR descriptor-name

Specifies an SQLDA descriptor that contains addresses and data types of input parameters or variables.

The descriptor must be a structure declared in the host language program as an SQLDA. If the program is precompiled and uses the embedded SQL statement `INCLUDE SQLDA`, the name of the structure is simply `SQLDA`. Programs can use multiple SQLDAs, but must explicitly declare them with names other than `SQLDA`.

Programs can always use the `USING DESCRIPTOR` clause of the `EXECUTE` statement whether or not the statement string contains input parameter markers, as long as the value of the `SQLD` field in the `SQLDA` corresponds to the number of input parameter markers. SQL updates the `SQLD` field with the correct number of input parameter markers when it processes the `DESCRIBE` statement for the statement string.

USING parameter

USING qualified-parameter

USING variable

Specifies input parameters or variables whose values SQL uses to replace parameter markers in the prepared statement string.

When you specify a list of parameters or variables, the number of parameters in the list must be the same as the number of input parameter markers in the statement string of the prepared statement. If SQL determines that a statement string had no input parameter markers, the `USING` clause is not allowed.

Usage Notes

- You must use at least one `USING` or one `INTO` clause in an `EXECUTE` statement. If the statement has no parameters then use the `EXECUTE IMMEDIATE` statement instead.
- You may mix parameters with `DESCRIPTOR` structures within the `EXECUTE` statement. That is, you may use `INTO DESCRIPTOR` to hold the results of the dynamic statement, but use `USING` with a list of parameters to provide the input values.
- When you issue the `EXECUTE` statement for a previously prepared statement, you might want to obtain information beyond the success or failure code returned in the `SQLCODE` status parameter. For example, you might want to know how many rows were affected by the execution

EXECUTE Statement

of an INSERT, DELETE, UPDATE, FETCH, or SELECT statement. SQL returns this information in the SQLERRD[2] field of the SQLCA.

However, when you use an SQLCA parameter to prepare a statement, you must also use an SQLCA parameter when you execute that statement. For example, using SQL module language calls from C, your code might look like the following where the SQLCA parameter is passed to both procedures:

```
static struct SQLCA sqlca;
/* ... */
PREPARE_STMT(&sqlca, statement, &stmt_id);
/* ... */
EXECUTE_STMT(&sqlca, &stmt_id);
```

For more information about the SQLCA, including the SQLERRD[2] field, see Appendix C.

Example

Example 1: Executing an INSERT statement with parameter markers

These fragments from the online sample C program `sql_dynamic` illustrate using an EXECUTE statement in an SQL module procedure to execute a dynamically generated SQL statement.

The program accepts input of any valid SQL statement from the terminal and calls the subunit shown in the following program excerpt:

```
.
.
.
/*
**-----
** Begin Main routine
**-----
*/

int sql_dynamic (psql_stmt, input_sqlda, output_sqlda, stmt_id, is_select)
char *psql_stmt;
sqlda *input_sqlda;
sqlda *output_sqlda;
long *stmt_id;
int *is_select;

{
    sqlda sqlda_in, sqlda_out; /* Declare the SQLDA structures. */
    int rowcount, status;
    int param;

/* Declare arrays for storage of original data types and allocate memory. */
```

EXECUTE Statement

```
mem_ptr output_save;
mem_ptr input_save;

/* * If a NULL SQLDA is passed, then a new statement is being prepared. */
if ((*input_sqlda == NULL) && (*output_sqlda == NULL))
{
    new_statement = TRUE;

    /*
    * Allocate separate SQLDAs for input parameter markers (SQLDA_IN)
    * and output list items (SQLDA_OUT). Assign the value of the constant
    * MAXPARAMS to the SQLN field of both SQLDA structures. SQLN specifies to
    * SQL the maximum size of the SQLDA.
    */

    if ((sqlda_in = (sqlda) calloc (1, sizeof (sqlda_rec))) == 0)
    {
        printf ("\n\n*** Error allocating memory for sqlda_in: Abort");
        return (-1);
    }
    else /* set # of possible parameters */
        sqlda_in->sqln = MAXPARAMS;

    if ((sqlda_out = (sqlda) calloc (1, sizeof (sqlda_rec))) == 0)
    {
        printf ("\n\n*** Error allocating memory for sqlda_out: Abort");
        return (-1);
    }
    else
        /* Set # of possible select list items. */
        sqlda_out->sqln = MAXPARAMS;

    /* copy name SQLDA2 to identify the SQLDA */
    strncpy(&sqlda_in->sqlda_id[0], "SQLDA2 ", 8);
    strncpy(&sqlda_out->sqlda_id[0], "SQLDA2 ", 8);

    /*
    * Call an SQL module language procedure, prepare_stmt and
    * describe_stmt that contains a PREPARE and DESCRIBE...OUTPUT
    * statement to prepare the dynamic statement and write information
    * about any select list items in it to SQLDA_OUT.
    */

    *stmt_id = 0; /* If <> 0 the BADPREPARE error results in the PREPARE.*/
```

EXECUTE Statement

```
PREPARE_STMT (&SQLCA, stmt_id, psql_stmt);
if (SQLCA.SQLCODE != sql_success)
{
    printf ("\n\nDSQL-E-PREPARE, Error %d encountered in PREPARE",
            SQLCA.SQLCODE);
    display_error_message();
    return (-1);
}

DESCRIBE_SELECT (&SQLCA, stmt_id, sqlda_out);
if (SQLCA.SQLCODE != sql_success)
{
    printf ("\n\nDSQL-E-PREPARE, Error %d encountered in PREPARE",
            SQLCA.SQLCODE);
    display_error_message();
    return (-1);
}

/*
 * Call an SQL module language procedure, describe_parm, that contains a
 * DESCRIBE...INPUT statement to write information about any parameter
 * markers in the dynamic statement to sqlda_in.
 */

DESCRIBE_PARM (&SQLCA, stmt_id, sqlda_in);
if (SQLCA.SQLCODE != sql_success)
{
    printf ("\n\n*** Error %d returned from describe_parm: Abort",
            SQLCA.SQLCODE);
    display_error_message();
    return (-1);
}

/* Save the value of the SQLCA.SQLERRD[1] field so that program can
 * determine if the statement is a SELECT statement or not.
 * If the value is 1, the statement is a SELECT statement.*/
*is_select = SQLCA.SQLERRD[1];
.
.
.

/*
 * Check to see if the prepared dynamic statement contains any parameter
 * markers by looking at the SQLD field of sqlda_in. SQLD contains the
 * number of parameter markers in the prepared statement. If SQLD is
 * positive, the prepared statement contains parameter markers. The program
 * executes a local procedure, get_in_params, that prompts the user for
 * values, allocates storage for those values, and updates the SQLDATA field
 * of sqlda_in:
 */
```

EXECUTE Statement

```
if (sqlda_in->sqld > 0)
    if ((status = get_in_params(sqlda_in,input_save)) != 0)
        {
            printf ("\nError returned from GET_IN_PARAMS. Abort");
            return (-1);
        }

/* Check to see if the prepared dynamic statement is a SELECT by looking
* at the value in is_select, which stores the value of the
* SQLCA.SQLERRD[1] field. If that value is equal to 1, the prepared
* statement is a SELECT statement. The program allocates storage for
* rows for SQL module language procedures to open and fetch from a cursor,
* and displays the rows on the terminal:
*/

if (*is_select)
    {
        if (new_statement == TRUE)      /* Allocate buffers for output. */
            {
                /* assign a unique name for the cursor */
                sprintf(cursor_name,"%2d",++cursor_counter);

                if ((status = allocate_buffers(sqlda_out)) != 0)
                    .
                    .
                    .

/*
* If the SQLCA.SQLERRD[1] field is not 1, then the prepared statement is not a
* SELECT statement and only needs to be executed. Call an SQL module language
* procedure to execute the statement, using information about parameter
* markers stored in sqlda_in by the local procedure get_in_params:
*/
                {
                    EXECUTE_STMT (&SQLCA, stmt_id, sqlda_in);
                    if (SQLCA.SQLCODE != sql_success)
                        .
                        .
                        .

The SQL module language procedures called by the preceding fragment:

.
.
.

-----
-- Procedure Section
-----

-- This procedure prepares a statement for dynamic execution from the string
-- passed to it. It also writes information about the number and data type of
-- any select list items in the statement to an SQLDA2 (specifically,
-- the sqlda_out SQLDA2 passed to the procedure by the calling program).
--
```


EXECUTE Statement

```
PROCEDURE PREPARE_STMT
  SQLCA
  :DYN_STMT_ID      INTEGER
  :STMT              CHAR(1024);

  PREPARE :DYN_STMT_ID FROM :STMT;

-- This procedure writes information to an SQLDA (specifically,
-- the sqlda_in SQLDA passed to the procedure by the calling program)
-- about the number and data type of any parameter markers in the
-- prepared dynamic statement. Note that SELECT statements may also
-- have parameter markers.

PROCEDURE DESCRIBE_SELECT
  SQLCA
  :DYN_STMT_ID      INTEGER
  SQLDA;

  DESCRIBE :DYN_STMT_ID OUTPUT INTO SQLDA;

PROCEDURE DESCRIBE_PARM
  SQLCA
  :DYN_STMT_ID      INTEGER
  SQLDA;

  DESCRIBE :DYN_STMT_ID INPUT INTO SQLDA;

-- This procedure dynamically executes a non-SELECT statement.
-- SELECT statements are processed by DECLARE CURSOR, OPEN CURSOR,
-- and FETCH statements.
--
-- The EXECUTE statement specifies an SQLDA2 (specifically,
-- the sqlda_in SQLDA2 passed to the procedure by the calling program)
-- as the source of addresses for any parameter markers in the dynamic
-- statement.
--
-- The EXECUTE statement with the USING DESCRIPTOR clause
-- also handles statement strings that contain no parameter markers.
-- If a statement string contains no parameter markers, SQL sets
-- the SQLD field of the SQLDA2 to zero.

PROCEDURE EXECUTE_STMT
  SQLCA
  :DYN_STMT_ID      INTEGER
  SQLDA;

  EXECUTE :DYN_STMT_ID USING DESCRIPTOR SQLDA;
  .
  .
  .
```

EXECUTE IMMEDIATE Statement

EXECUTE IMMEDIATE Statement

Dynamically prepares, executes, and releases an SQL statement.

The EXECUTE IMMEDIATE statement is a dynamic SQL statement. Dynamic SQL lets programs accept or generate SQL statements at run time, in contrast to precompiled statements, which must be embedded in the program before it is compiled. Unlike embedded statements, such dynamically executed SQL statements are not necessarily part of the program's source code, but can be created while the program is running. Dynamic SQL is useful when you cannot predict the type of SQL statement your program will need to process.

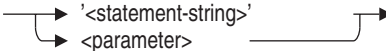
The EXECUTE IMMEDIATE statement cannot contain parameter markers. However, if the statement meets those restrictions and will be dynamically executed only once, use the EXECUTE IMMEDIATE statement instead of PREPARE and EXECUTE statements.

Environment

You can use the EXECUTE IMMEDIATE statement:

- Embedded in host language programs to be precompiled
- As part of a procedure in an SQL module

Format

`EXECUTE IMMEDIATE` 

Arguments

statement-string

parameter

Specifies the SQL statement to be prepared and executed dynamically. You either specify the statement string directly in a character string literal enclosed in single quotation marks, or in a parameter that contains the statement string.

EXECUTE IMMEDIATE Statement

Whether specified directly or by a parameter, the statement string must be a character string that is a dynamically executable SQL statement other than the SELECT statement. The form for the statement is the same as in embedded SQL, except that you do not need to begin the string with EXEC SQL or end it with any statement terminator.

Example

Example 1: Executing an INSERT statement with the EXECUTE IMMEDIATE statement

This COBOL program illustrates using the EXECUTE IMMEDIATE statement to prepare and execute a dynamic INSERT statement. Compare this example with the example for the EXECUTE statement (see the EXECUTE Statement), which uses an INSERT statement with parameter markers and displays the result of the insert operation.

```
IDENTIFICATION DIVISION.
PROGRAM-ID. EXECUTE_IMMEDIATE_EXAMPLE.
*
* Illustrate EXECUTE_IMMEDIATE with a dynamic INSERT statement.
*
DATA DIVISION.

WORKING-STORAGE SECTION.

* Variable for DECLARE SCHEMA:
01 FILESPEC PIC X(20).

* Variables to hold values for
* storage in EMPLOYEES:
01 EMP_ID PIC X(5).
01 FNAME PIC X(10).
01 MID_INIT PIC X(1).
01 LNAME PIC X(14).
01 ADDR_1 PIC X(25).
01 ADDR_2 PIC X(25).
01 CITY PIC X(20).
01 STATE PIC X(2).
01 P_CODE PIC X(5).
01 SEX PIC X(1).
01 BDATE PIC S9(11)V9(7) COMP.
01 S_CODE PIC X(1).
```

EXECUTE IMMEDIATE Statement

```
* Indicator variables for retrieving
* the entire row, including columns we
* do not assign values to, from
* the EMPLOYEES table:
01 EMP_ID_IND    PIC S9(4) COMP.
01 FNAME_IND    PIC S9(4) COMP.
01 MID_INIT_IND PIC S9(4) COMP.
01 LNAME_IND    PIC S9(4) COMP.
01 ADDR_1_IND   PIC S9(4) COMP.
01 ADDR_2_IND   PIC S9(4) COMP.
01 CITY_IND     PIC S9(4) COMP.
01 STATE_IND    PIC S9(4) COMP.
01 P_CODE_IND   PIC S9(4) COMP.
01 SEX_IND      PIC S9(4) COMP.
01 BDATE_IND    PIC S9(4) COMP.
01 S_CODE_IND   PIC S9(4) COMP.

* Buffer for error handling:
01 BUFFER       PIC X(300).
01 LEN         PIC S9(4) USAGE IS COMP.

* 01 disp_sqlcode      pic s9(9) sign leading separate.

* Load definition for SQL Communication Area (SQLCA):
EXEC SQL      INCLUDE SQLCA END-EXEC.

*****
*
*           P R O C E D U R E   D I V I S I O N
*
*****
PROCEDURE DIVISION.
START-UP.

* Assign value to FILESPEC:
      MOVE "SQL$DATABASE" TO FILESPEC

* Declare the schema:
      EXEC SQL DECLARE SCHEMA RUNTIME FILENAME :FILESPEC
      END-EXEC

*           Use an EXECUTE IMMEDIATE statement
*           to execute an INSERT statement:
      EXEC SQL EXECUTE IMMEDIATE
      "INSERT INTO EMPLOYEES
-         "(EMPLOYEE_ID,FIRST_NAME, LAST_NAME,CITY)
-         "VALUES ('99999','Les','Warton','Hudson')"
      END-EXEC
      PERFORM CHECK.

      PERFORM FETCHES.

EXEC SQL EXECUTE IMMEDIATE 'ROLLBACK' END-EXEC.
      PERFORM CHECK.

      DISPLAY "Rolled back changes. All done."
```

EXECUTE IMMEDIATE Statement

```
CLEAR-IT-EXIT.  
    EXIT PROGRAM.  
  
FETCHES.  
    DISPLAY "Here's the row we stored:"  
  
    EXEC SQL PREPARE STMT FROM  
    'SELECT * FROM EMPLOYEES WHERE EMPLOYEE_ID = "99999"'  
    END-EXEC  
    EXEC SQL DECLARE C CURSOR FOR STMT END-EXEC  
  
    EXEC SQL OPEN C END-EXEC  
*   Clear values in host language  
*   variables in case new values  
*   from the table are null:  
    MOVE SPACES TO EMP_ID  
    MOVE SPACES TO FNAME  
    MOVE SPACES TO MID_INIT  
    MOVE SPACES TO LNAME  
    MOVE SPACES TO ADDR_1  
    MOVE SPACES TO ADDR_2  
    MOVE SPACES TO CITY  
    MOVE SPACES TO STATE  
    MOVE SPACES TO P_CODE  
    MOVE SPACES TO SEX  
    MOVE ZERO TO BDATE  
    MOVE SPACES TO S_CODE  
  
    EXEC SQL FETCH C INTO  
        :EMP_ID:EMP_ID_IND,  
        :LNAME:LNAME_IND,  
        :FNAME:FNAME_IND,  
        :MID_INIT:MID_INIT_IND,  
        :ADDR_1:ADDR_1_IND,  
        :ADDR_2:ADDR_2_IND,  
        :CITY:CITY_IND,  
        :STATE:STATE_IND,  
        :P_CODE:P_CODE_IND,  
        :SEX:SEX_IND,  
        :BDATE:BDATE_IND,  
        :S_CODE:S_CODE_IND  
    END-EXEC
```

EXECUTE IMMEDIATE Statement

```
        DISPLAY EMP_ID, " ",
            FNAME, " ",
            MID_INIT, " ",
            LNAME, " ",
            ADDR_1, " ",
            ADDR_2, " ",
            CITY, " ",
            STATE, " ",
            P_CODE, " ",
            SEX, " ",
            BDATE, " ",
            S_CODE.

    PERFORM CHECK.
    EXEC SQL CLOSE C END-EXEC.

CHECK.
    IF SQLCODE NOT = 100 AND SQLCODE NOT = 0
        DISPLAY "Error: SQLCODE = ", SQLCODE
        CALL "SQL$GET_ERROR_TEXT" USING
            BY DESCRIPTOR BUFFER,
            BY REFERENCE LEN
        DISPLAY BUFFER(1:LEN)
    END-IF.
```

EXIT Statement

Stops an interactive SQL session and returns you to the operating system prompt. By default, the EXIT statement commits changes made during the session.

Environment

You can issue the EXIT statement in interactive SQL only.

Format

→ EXIT →
→ <CTRL/Z> →

Usage Notes

- Both the QUIT and EXIT statements end an interactive SQL session. The QUIT statement automatically rolls back changes made during the session; the EXIT statement, by default, commits changes made during the session.
- If you have made uncommitted changes to the database when you issue the EXIT statement, SQL asks if you want to roll back the transaction.

There are uncommitted changes to this database.
Would you like a chance to ROLLBACK these changes (No)?

If you do not answer and press the Return key or type NO, SQL commits all changes made since the last COMMIT or ROLLBACK statement. If you answer YES to the prompt, SQL returns you to the SQL prompt.

- Typing Ctrl/Z is the same as issuing the EXIT statement for OpenVMS.

EXPORT Statement

EXPORT Statement

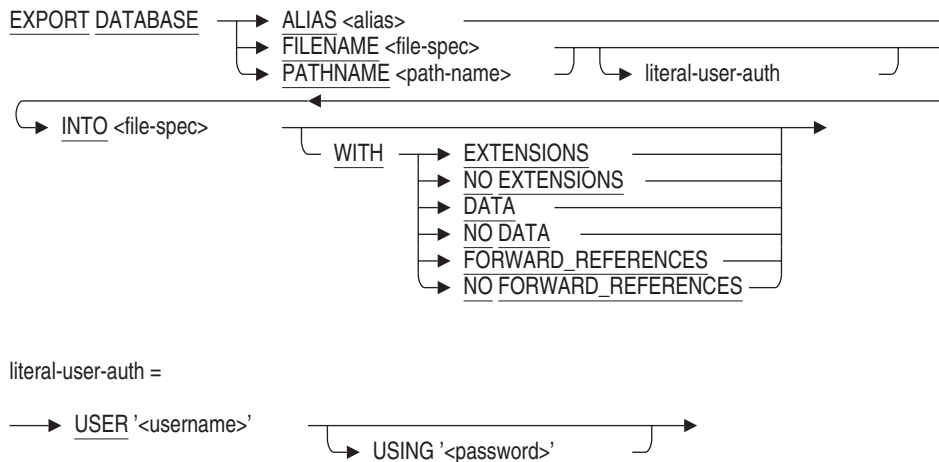
Makes a copy of a database in an intermediate form. Use the IMPORT statement to rebuild an Oracle Rdb database from the interchange file (.rbr file extension) created by the EXPORT statement.

You use the EXPORT statement with the IMPORT statement to make changes to Oracle Rdb databases that cannot be made any other way. The EXPORT statement unloads a database to an .rbr file. The IMPORT statement creates the database again with the changes that are both allowed and not allowed through ALTER statements. See the IMPORT Statement for more information.

Environment

You can use the EXPORT statement in interactive SQL only.

Format



Arguments

ALIAS alias

FILENAME file-spec

PATHNAME path-name

Specifies the source database files to be written to an .rbr file.

EXPORT Statement

- The **ALIAS** argument specifies the alias of an already attached database. If the database you want to export is already attached, specifying **ALIAS** avoids the overhead of a second attach to the database and the locking that attach entails.
- The **FILENAME** and **PATHNAME** arguments both identify the database root file associated with the database. If you specify a repository path name, the path name indirectly specifies the database root file. Because the **EXPORT** statement does not change any definitions in the repository, the effect of the **PATHNAME** and **FILENAME** arguments is the same.

FORWARD_REFERENCES

NO FORWARD_REFERENCES

The **EXPORT** statement analyzes all dependencies in the database to determine which functions and procedures are referenced by other definitions. Since **IMPORT** defines each object type in a strict order, it is possible that some definitions may be used prior to their definition. For instance, tables are defined before modules, but the table might call an SQL function from a module. The **FORWARD_REFERENCES** option requests that **EXPORT** save descriptions of these routines first in the interchange file so that **IMPORT** can declare them prior to their usage. See the **DECLARE Routine Statement** for more details.

FORWARD_REFERENCES is the default. If the interchange file is to be used by a version prior to Oracle Rdb V7.1.0.4 then the **NO FORWARD_REFERENCES** option should be used to exclude this information.

INTO file-spec

Specifies the name for the **.rbr** file the **EXPORT** statement creates. Optionally, the file specification can include a device and directory specification.

literal-user-auth

Specifies the user name and password for access to databases, particularly remote database.

This literal lets you explicitly provide user name and password information in the **EXPORT** statement.

USER 'username'

Defines a character string literal that specifies the operating system user name that the database system uses for privilege checking.

USING 'password'

Defines a character string literal that specifies the user's password for the user name specified in the **USER** clause.

EXPORT Statement

WITH DATA

WITH NO DATA

Specifies whether the .rbr file created by the EXPORT statement includes the data and metadata contained in the database, or the metadata only. The default is WITH DATA.

When you specify the WITH NO DATA option, the EXPORT statement copies metadata, but not the data, from a source database to an .rbr file. Use the IMPORT statement to generate an empty database whose metadata is identical to that of the source database.

Note

The WITH NO DATA option is not compatible with Oracle CDD/Repository databases (CDD\$DATABASE.RDB). If you attempt to export a CDD\$DATABASE.RDB database, SQL issues an error message stating that the WITH NO DATA option is not valid for Oracle CDD/Repository databases.

WITH EXTENSIONS

WITH NO EXTENSIONS

Specifies whether or not the .rbr file created by the EXPORT statement includes extensions that are compatible only with Oracle Rdb Version 3.0 or higher database systems. The default is WITH EXTENSIONS.

When you specify the WITH NO EXTENSIONS option, the resulting interchange (.rbr) file contains only the definitions of the domains, the tables, and indexes. Indexes are converted to sorted indexes and are minus storage maps. The following conversions take place for domains:

- TINYINT data types are converted to SMALLINT data types
- DATE ANSI, TIMESTAMP, and TIME data types are converted to DATE VMS data types

In addition, all null values are converted to the columns' missing value or default to a data type specific missing value. For example, null numeric values are replaced by zeros and null character values are replaced by blanks.

When you specify the WITH NO EXTENSIONS option, many features of Oracle Rdb databases are not exported. For example, storage areas, storage maps, triggers, collating sequences, functions, modules, and outlines are not backed up when you specify the WITH NO EXTENSIONS argument.

EXPORT Statement

Note

The WITH NO EXTENSIONS option is not compatible with Oracle CDD/Repository databases (CDD\$DATABASE.RDB). If you attempt to export a CDD\$DATABASE.RDB database, SQL issues an error message stating that the WITH NO EXTENSIONS option is not valid for Oracle CDD/Repository databases.

Usage Notes

- Before using EXPORT and IMPORT statements, be sure that the database was backed up in case either the EXPORT or IMPORT statement fails. Use the RMU Backup command.
- For information about how SQL handles character set information when you export and import a database, see the IMPORT Statement.
- You cannot use the EXPORT statement on a database that has over 65,535 segments in one segmented string field. However, the RMU Backup command does not have this limitation on the number of segments within a segmented string.
- The EXPORT statement does not check for a corrupt database. If the database is corrupt, EXPORT and IMPORT statements create a valid database, but the contents of that database may not be identical to the original.
- You need read access to all the objects in the database to back up the database with the EXPORT statement.
- If you use the ALTER DATABASE statement to set OPEN IS MANUAL on a database, you cannot export that database if it is closed.
- See the *Oracle Rdb Guide to Database Maintenance* for a complete discussion of when to use the IMPORT, EXPORT, and ALTER DATABASE statements.
- It is not possible to export a database using the WITH NO EXTENSIONS clause if it contains INTERVAL domains. Oracle Rdb recommends either removing the offending domains and related columns (see the DROP DOMAIN Statement) or performing the EXPORT operation without including the WITH NO EXTENSIONS clause.

EXPORT Statement

- Normally, during an export operation, the Oracle Rdb interchange file (.rbr), which uses the Record Management Services (RMS) default extent, will extend for every 3 blocks the .rbr file grows in size. To prevent this, define the following SET statement to change the process default RMS extent quantity:

```
$ SET RMS_DEFAULT/EXTEND_QUANTITY=30000
```

Now, rather than “extending” the .rbr file for every 3 blocks (which involves many extend operations), the RMS extend is only invoked *once* per 30,000 blocks. By specifying a larger value for the file extend parameter, the run time of the export operation can be significantly improved.

- Oracle Rdb does not support remote export between different versions of Oracle Rdb. You can successfully export a database only if the version number of the system from which you issue the EXPORT statement equals the version number of the database you are exporting.
- A node specification may be specified for the root FILENAME clause of the EXPORT DATABASE statement.

FETCH Statement

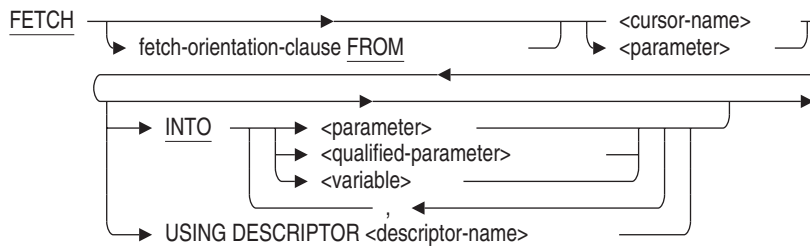
Advances a cursor to the next row of its result table and retrieves the values from that row. When used with a list cursor, the FETCH statement places the cursor on a specified position within a list and retrieves a portion of that list. When embedded in precompiled host language programs, the FETCH statement assigns the values from the row to host parameters. In interactive SQL, the FETCH statement displays the value of the row on the terminal screen.

Environment

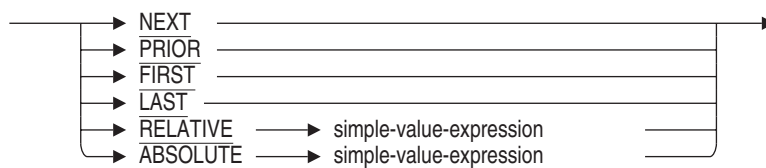
You can use the FETCH statement:

- In interactive SQL (except for the USING DESCRIPTION clause)
- Embedded in host language programs to be precompiled
- As part of a procedure in an SQL module

Format



fetch-orientation-clause =



FETCH Statement

Arguments

cursor-name

parameter

Specifies the name of the cursor from which you want to retrieve a row. Use a parameter if the cursor referred to by the cursor name was declared at run time with a dynamic DECLARE CURSOR statement. Specify the parameter used for the cursor name in the dynamic DECLARE CURSOR statement.

You can use a parameter to refer to the cursor name only when the FETCH statement is accessing a dynamic cursor.

fetch-orientation-clause FROM

Specifies the specific segment of the list cursor to fetch. These options are available only if you specified the SCROLL option in the DECLARE CURSOR statement. The choices are:

- **NEXT**
Fetches the next segment of the list cursor. This is the default.
- **PRIOR**
Fetches the segment immediately before the current segment of the list cursor.
- **FIRST**
Fetches the first segment of the list cursor.
- **LAST**
Fetches the last segment of the list cursor.
- **RELATIVE simple-value-expression**
Fetches the segment of the list cursor indicated by the value expression. For example, relative -4 would fetch the segment that is four segments prior to the current segment.
- **ABSOLUTE simple-value-expression**
Fetches the segment of the list cursor indicated by the value expression. For example, absolute 4 would fetch the fourth segment of the list cursor.

INTO parameter

INTO qualified-parameter

INTO variable

Specifies a list of parameters, qualified parameters (host structures), or variables to receive the values SQL retrieves from the row of the cursor's result table. The number of parameters or variables in the list must be the same as

FETCH Statement

the number of values in the row. (If any of the parameters is a host structure, SQL counts the number of parameters in that structure when it compares the number of host parameters in the INTO clause with the number of values in the row.)

The data types of parameters and variables must be compatible with the values of the corresponding column of the cursor row.

simple-value-expression

Specifies either a positive or negative integer, or a numeric module language or host language parameter.

USING DESCRIPTOR descriptor-name

Specifies the name of a descriptor that corresponds to an SQLDA. If you use the INCLUDE statement to insert the SQLDA into your program, the descriptor name is simply SQLDA.

An SQLDA is a collection of host language variables used only in dynamic SQL. In a FETCH statement, the SQLDA points to a number of parameters SQL uses to store values from the row. The number of parameters must match the number of columns in the row.

The data types of parameters must be compatible with the values of the corresponding column of the cursor row.

Usage Notes

- You cannot use a FETCH statement for a cursor before you issue an OPEN statement for that cursor.
- An open cursor can be positioned:
 - Before the first row of its result table. When an OPEN statement executes, SQL positions the cursor before the first row. When a DELETE statement that refers to a cursor executes, SQL positions the cursor before the next row that follows the deleted row.
 - On a row of its result table (after a FETCH statement for any but the last row).
 - After the last row of its result table.

When the table cursor is positioned on the last row, any FETCH or DELETE statement from the cursor positions the cursor after the last row.

FETCH Statement

- An error is generated and the SQLCODE status parameter or SQLCODE field of SQLCA is set to +100 and the SQLSTATE field is set to '02000' in the following situations:
 - If the current position of a cursor in a FETCH or FETCH NEXT statement is on or after the last row of its result table.
 - If a FETCH ABSOLUTE or FETCH RELATIVE statement tries to retrieve rows that are out of range.
 - If the current position of a cursor in a FETCH PRIOR statement is on or before the first row of its result table.
- If you attempt to fetch an element of a list into a target specification that is shorter than the element, the element will be truncated. The sixth element of the SQLERRD array of the SQLCA is set to the difference between the element and the target (the number of truncated bytes).
- Always use an indicator array when you use host language structures. For information about indicator arrays, see Section 2.2.13.2 or the *Oracle Rdb Guide to SQL Programming*.

When SQL fetches a list cursor, the value of the indicator parameter shows if the segment is truncated. If no truncation occurs, the value of the indicator parameter is 0. If the list segment value is null, the value of the indicator parameter is -1. If the list segment is truncated, the SQLLEN stores the length of the untruncated segment.
- You can determine the length of the fetched segment by passing a VARCHAR or VARBYTE field in the SQLDA for the segment. SQL returns the length of the segment in the length field of these two data types.
- You must make sure you close the list cursor before fetching the next row of a table cursor. SQL does not issue an error message or warning if you forget to do so.

See Appendix C for more information on the SQLCA and SQLSTATE.

Examples

Example 1: Using a FETCH statement embedded in a PL/I program

This program fragment uses embedded DECLARE CURSOR, OPEN, and FETCH statements to retrieve and print the names and departments of managers. The FETCH statement fetches the rows of the result table and stores them in the parameters :FNAME, :LNAME, and :DNAME.

FETCH Statement

```
/* Declare the parameters: */
BEGIN DECLARE SECTION

DCL ID          CHAR(3);
DCL FNAME       CHAR(10);
DCL LNAME       CHAR(14);

END DECLARE SECTION

/* Declare the cursor: */
EXEC SQL DECLARE MANAGER CURSOR FOR
    SELECT E.FIRST_NAME, E.LAST_NAME, D.DEPARTMENT_NAME
    FROM EMPLOYEES E, DEPARTMENTS D
    WHERE E.EMPLOYEE_ID = D.MANAGER_ID ;

/* Open the cursor: */
EXEC SQL OPEN MANAGER;

/* Start a loop to process the rows of the cursor: */
DO WHILE (SQLCODE = 0);
    /* Retrieve the rows of the cursor
    and put the value in parameters: */
    EXEC SQL FETCH MANAGER INTO :FNAME, :LNAME, :DNAME;
    /* Print the values in the parameters: */
        .
        .
        .
END;

/* Close the cursor: */
EXEC SQL CLOSE MANAGER;
```

Example 2: Using a FETCH statement to display segments in a column of data type LIST

This interactive example uses a table cursor to retrieve a row that contains a list from the RESUMES table. The OPEN statement positions the cursor on the first segment of the list in the RESUME column, and subsequent FETCH statements retrieve successive segments of that list.

FETCH Statement

```
SQL> DECLARE TBLCURSOR2 CURSOR FOR SELECT EMPLOYEE_ID, RESUME
cont> FROM RESUMES;
SQL> DECLARE LSTCURSOR2 LIST CURSOR FOR SELECT RESUME
cont> WHERE CURRENT OF TBLCURSOR2;
SQL> OPEN TBLCURSOR2;
SQL> FETCH TBLCURSOR2;
    00164
SQL> OPEN LSTCURSOR2;
SQL> FETCH LSTCURSOR2;
    RESUME
    This is the resume for 00164
SQL> FETCH LSTCURSOR2;
    RESUME
    Boston, MA
SQL> FETCH LSTCURSOR2;
    RESUME
    Oracle Corporation
SQL> FETCH LSTCURSOR2;
    RESUME
%RDB-E-STREAM_EOF, attempt to fetch past end of record stream
SQL> CLOSE LSTCURSOR2;
SQL> SELECT * FROM RESUMES;
    EMPLOYEE_ID    RESUME
    00164                72:2:3
1 row selected
SQL> CLOSE TBLCURSOR2;
SQL> COMMIT;
```

Example 3: Using a scrollable list cursor to fetch list data

This C program demonstrates the use of scrollable list cursors to read list data from the sample personnel database using the `FETCH` statement. The list data being read is from the `RESUME` column of the `RESUMES` table in personnel. Note that the `RESUME` is divided into three segments in this order:

1. A line including the employee's name: "This is the resume for Alvin Toliver"
2. A line stating where the employee lives: "Boston, MA"
3. A line stating where the employee works: "Oracle Corporation"

```
#include stdio
#include descrip

/* Declare parameters for error handling by including the SQLCA. */
EXEC SQL INCLUDE SQLCA;

/* Error-handling section. */
dump_error( )
{
```

FETCH Statement

```
short          errbuflen;
char           errbuf[ 1024 ];
struct         dsc$descriptor_s  errbufdsc;

errbufdsc.dsc$b_class = DSC$K_CLASS_S;
errbufdsc.dsc$b_dtype = DSC$K_DTYPE_T;
errbufdsc.dsc$w_length = 1024;
errbufdsc.dsc$a_pointer = &errbuf;

    if (SQLCA.SQLCODE != 0)
    {
        printf( "SQLCODE = %d\n", SQLCA.SQLCODE );
        SQL$GET_ERROR_TEXT( &errbufdsc, &errbuflen );
        errbuf[ errbuflen ] = 0;
        printf("%s\n", &errbuf );
    }
}
main()
{
    /* Attach to the personnel database. */
    EXEC SQL DECLARE ALIAS FILENAME personnel;
    /* Declare variables. */
    short two_s;
    long  two_l;
    char  blob[8];
    char  emp_id[6];
    char  seg2[ 81 ];

    /* Declare a table cursor. */
        exec sql declare resumes_cursor table cursor for
        select employee_id, resume from resumes where employee_id = '00164';
    /* Declare a read-only scrollable list cursor to fetch the RESUME column. */
        exec sql declare resume_list_cursor read only scrollable list cursor for
        select resume where current of resumes_cursor;
    /* Open the table cursor. */
        exec sql open resumes_cursor;
        dump_error();

    /* Place the first value in the table cursor (00164) into the emp_id parameter,
    and the resume data into the blob parameter. */
        exec sql fetch resumes_cursor into :emp_id, :blob;
        dump_error();

    /* Open the scrollable list cursor. */
    exec sql open resume_list_cursor;
    dump_error();
```

FETCH Statement

```
/* Begin to use the FETCH statement to read desired lines from the resume.
   If an attempt is made to retrieve a segment that is out of range, the
   program prints an error message.
*/

exec sql fetch last from resume_list_cursor into :seg2;
printf("FETCH LAST segment returned: %s\n", seg2 );
dump_error();

exec sql fetch next from resume_list_cursor into :seg2;
printf("FETCH NEXT segment returned: %s\n", seg2 );
dump_error();

exec sql fetch first from resume_list_cursor into :seg2;
printf("FETCH FIRST segment returned: %s\n", seg2 );
dump_error();

exec sql fetch next from resume_list_cursor into :seg2;
printf("FETCH NEXT segment returned: %s\n", seg2 );
dump_error();

exec sql fetch next from resume_list_cursor into :seg2;
printf("FETCH NEXT segment returned: %s\n", seg2 );
dump_error();

exec sql fetch relative -2 from resume_list_cursor into :seg2;
printf("FETCH RELATIVE -2 segment returned: %s\n", seg2 );
dump_error();

exec sql fetch first from resume_list_cursor into :seg2;
printf("FETCH FIRST segment returned: %s\n", seg2 );
dump_error();

exec sql fetch relative 2 from resume_list_cursor into :seg2;
printf("FETCH RELATIVE 2 segment returned: %s\n", seg2 );
dump_error();

exec sql fetch last from resume_list_cursor into :seg2;
printf("FETCH LAST segment returned: %s\n", seg2 );
dump_error();

exec sql fetch prior from resume_list_cursor into :seg2;
printf("FETCH PRIOR segment returned: %s\n", seg2 );
dump_error();

exec sql fetch ABSOLUTE 1 from resume_list_cursor into :seg2;
printf("FETCH ABSOLUTE 1 segment returned: %s\n", seg2 );
dump_error();

exec sql fetch relative 2 from resume_list_cursor into :seg2;
printf("FETCH RELATIVE 2 segment returned: %s\n", seg2 );
dump_error();

two_s = 2;
exec sql fetch ABSOLUTE :two_s from resume_list_cursor into :seg2;
printf("FETCH ABSOLUTE :two_s segment returned: %s\n", seg2 );
dump_error();
```

FETCH Statement

```
two_1 = 2;
exec sql fetch ABSOLUTE :two_1 from resume_list_cursor into :seg2;
printf("FETCH ABSOLUTE :two_1 segment returned: %s\n", seg2 );
dump_error();

    exec sql fetch RELATIVE :two_1 from resume_list_cursor into :seg2;
    printf("FETCH RELATIVE :two_1 segment returned: %s\n", seg2 );
    dump_error();

    exec sql rollback;
}
```

The following example shows the output from the program:

```
FETCH LAST segment returned: Oracle Corporation
FETCH NEXT segment returned: Oracle Corporation
SQLCODE = 100
%SQL-W-NOTFOUND, No rows were found for this statement
FETCH FIRST segment returned: This is the resume for Alvin Toliver
FETCH NEXT segment returned: Boston, MA
FETCH NEXT segment returned: Oracle Corporation
FETCH RELATIVE -2 segment returned: This is the resume for Alvin Toliver
FETCH FIRST segment returned: This is the resume for Alvin Toliver
FETCH RELATIVE 2 segment returned: Oracle Corporation
FETCH LAST segment returned: Oracle Corporation
FETCH PRIOR segment returned: Boston, MA
FETCH ABSOLUTE 1 segment returned: This is the resume for Alvin Toliver
FETCH RELATIVE 2 segment returned: Oracle Corporation
FETCH ABSOLUTE :two_s segment returned: Boston, MA
FETCH ABSOLUTE :two_1 segment returned: Boston, MA
FETCH RELATIVE :two_1 segment returned: Boston, MA
SQLCODE = -1
%RDB-F-SEGSTR_EOF, attempt to fetch past the end of a segmented string
-RDMS-E-FETRELATIVE, fetch relative (2) causes reference out of range 1..3
```

FOR Control Statement

FOR Control Statement

Executes an SQL statement for each row of a query expression.

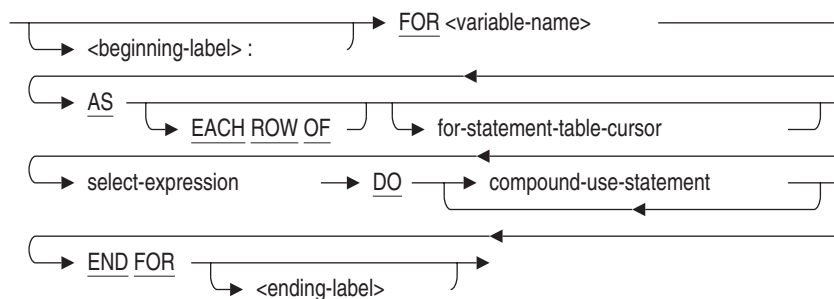
Environment

You can use the FOR control statement in a compound statement of a multistatement procedure:

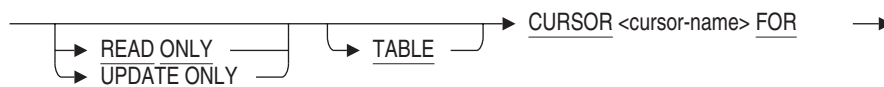
- In interactive SQL
- Embedded in host language programs to be precompiled
- As part of a procedure in an SQL module
- In dynamic SQL as a statement to be dynamically executed

Format

for-statement =



for-statement-table-cursor =



Arguments

AS EACH ROW OF for-statement-table-cursor

Creates a result table with a specified cursor.

The optional naming of a cursor lets you use positioned data manipulation language statements in the DO clause of a FOR loop.

FOR Control Statement

AS EACH ROW OF select-expression

Creates a simple result table.

After SQL creates the result table from the select expression, the DO clause executes a set of SQL statements (compound-use-statement) for each result table row.

beginning-label:

Assigns a name to the FOR statement.

A named FOR loop is called a **labeled FOR loop statement**. If you include an ending label, it must be identical to its corresponding beginning label. A beginning label must be unique within the procedure in which the label is contained.

DO compound-use-statement

Executes a block of SQL statements for each row of the select expression result table.

END FOR ending-label

Marks the end of a FOR loop. If you choose to include the optional ending label, it must match exactly its corresponding beginning label. An ending label must be unique within the procedure in which the label is contained.

The optional end-label argument makes the FOR loops of multistatement procedures easier to read, especially in very complex procedure blocks.

FOR variable-name

Specifies a name for a record consisting of a field for each named column of the FOR loop select expression. Each field in the record contains the data represented by each column name in each row of the select expression result table.

The variable name lets you reference a field in the compound-use-statement argument, for example: `variable-name.column-name`.

Usage Notes

- A beginning label used with the ITERATE statement lets you skip the commands of the loop body and start the next iteration of the loop.
- The cursor name must be unique within the containing module.
- Reference to the cursor name is only valid inside this FOR statement.

FOR Control Statement

- Variables are created at the beginning of the FOR statement and are destroyed at the end of the FOR statement.
- A FOR cursor loop executes the DO . . . END FOR body of the loop for each row fetched from the row set. Applications cannot use RETURNED_SQLCODE or RETURNED_SQLSTATE to determine if the FOR loop reached the end of the row set without processing any rows. Applications should use the GET DIAGNOSTICS ROW_COUNT statement after the END FOR clause to test for zero or more rows processed.

Examples

Example 1: Using the FOR statement within an SQL module procedure

```
SQL> set flags 'trace';
SQL>
SQL> create module REPORTS
cont> /*
***> This procedure counts the employees of a given state
***> who have had a decrease in their salary during their
***> employment
***> */
cont> procedure COUNT_DECREASED
cont>     (in :state CHAR(2)
cont>     ,inout :n_decreased INTEGER);
cont> begin
cont> set :n_decreased = 0;
cont>
cont> EMP_LOOP:
cont> for :empfor
cont>     as each row of
cont>         select employee_id
cont>         from EMPLOYEES where state = :state
cont> do
cont>     begin
cont>         declare :last_salary INTEGER (2) default 0;
cont>
cont>         HISTORY_LOOP:
cont>         for :salfor
cont>             as each row of
cont>                 select salary_amount
cont>                 from SALARY_HISTORY
cont>                 where employee_id = :empfor.employee_id
cont>                 order by salary_start
cont>         do
cont>             if :salfor.salary_amount < :last_salary
cont>             then
cont>                 set :n_decreased = :n_decreased + 1;
```


FOR Control Statement

```
cont>          trace :empfor.employee_id, ': ', :salfor.salary_amount;
cont>          leave HISTORY_LOOP;
cont>      end if;
cont>
cont>          set :last_salary = :salfor.salary_amount;
cont>      end for;
cont>      end;
cont> end for;
cont> end;
cont>
cont> end module;
SQL>
SQL> declare :n integer;
SQL> call COUNT_DECREASED ('NH', :n);
~Xt: 00200: 40789.00
~Xt: 00248: 46000.00
~Xt: 00471: 52000.00
      N
      3
SQL>
SQL> rollback;
```

FOR (Counted) Control Statement

FOR (Counted) Control Statement

Executes a block of SQL statements while the FOR loop variable is incremented (or decremented) from a user-specified starting value to a user-specified ending value.

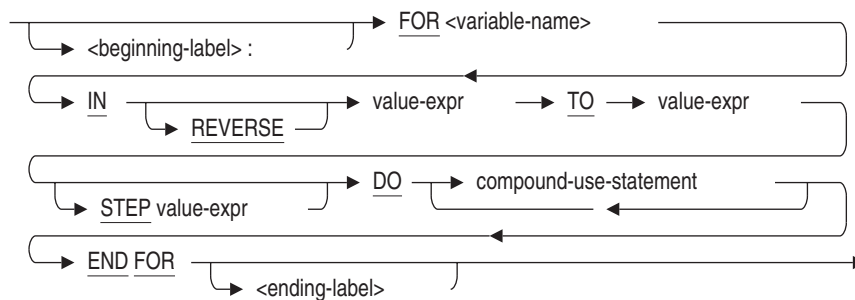
Environment

You can use the FOR counted control statement in a compound statement of a multistatement procedure:

- In interactive SQL
- Embedded in host language programs to be precompiled
- As part of a procedure in an SQL module
- In dynamic SQL as a statement to be dynamically executed

Format

counted-for-statement =



Arguments

AS EACH ROW OF select-expression

Creates a simple result table.

After SQL creates the result table from the select expression, the DO clause executes a set of SQL statements (compound-use-statement) for each result table row. See Section 2.8.1 for more information on select expressions.

FOR (Counted) Control Statement

beginning-label:

Assigns a name to the FOR statement. A named FOR loop is called a **labeled FOR loop statement**. If you include an ending label, it must be identical to its corresponding beginning label. A beginning label must be unique within the procedure in which the label is contained.

compound-use-statement

Identifies the SQL statements allowed in a compound statement block. See Compound Statement for a complete description of a compound statement.

DO compound-use-statement

Executes a block of SQL statements once for each execution of the loop as defined by the starting and ending value expressions.

END FOR

END FOR ending-label

Marks the end of a FOR loop. If you choose to include the optional ending label, it must match exactly its corresponding beginning label. An ending label must be unique within the procedure in which the label is contained. The optional end-label argument makes the FOR loops of multistatement procedures easier to read, especially in very complex procedure blocks.

FOR variable-name

Specifies a variable to hold a value that is incremented each time the FOR loop is executed. The variable is decremented if the REVERSE keyword is specified. The starting value for the variable is the first value expression. Execution of the FOR loop ends when the variable has been incremented (or decremented) to the value specified with the second value expression.

Marks the end of a FOR loop. If you choose to include the optional ending label, it must match exactly its corresponding beginning label. An ending label must be unique within the procedure in which the label is contained. The optional end-label argument makes the FOR loops of multistatement procedures easier to read, especially in very complex procedure blocks.

IN value-expr TO value-expr

IN REVERSE value-expr TO value-expr

Specifies how often the compound-use-statement should be executed. When the REVERSE keyword is not specified, the variable contained in the FOR variable-name is incremented at the end of each execution of the FOR loop body. When the REVERSE keyword is specified, the variable contained in the FOR variable-name is decremented at the end of each execution of the FOR loop body.

FOR (Counted) Control Statement

Both value expressions are evaluated once before the loop executes. The TO value-expression is evaluated first to ensure that references to the FOR loop variable do not cause side effects.

select expression

See Section 2.8.1 for a complete description of select expressions.

STEP value-expr

Controls the size of the increment between loop iterations. The step size is specified using a numeric value expression.

If omitted the default step size is 1.

```
SQL> begin
cont> declare :i integer;
cont> for :i in 1 to 20 step 5
cont> do
cont>     trace :i;
cont> end for;
cont> end;
~Xt: 1
~Xt: 6
~Xt: 11
~Xt: 16
```

Note

Even if the loop control variable is an INTERVAL type the STEP must be numeric type. In addition the value must be greater than zero - use the REVERSE keyword to decrement the loop control variable.

value-expr

Syntax:

```
IN value-expr TO value-expr
IN REVERSE value-expr TO value-expr
```

Specifies how often the compound-use-statement should be executed. When the REVERSE keyword is not specified, the variable contained in the FOR variable-name is incremented at the end of each execution of the FOR loop body. When the REVERSE keyword is specified, the variable contained in the FOR variable-name is decremented at the end of each execution of the FOR loop body.

FOR (Counted) Control Statement

Both value expressions are evaluated once before the loop executes. The TO value-expression is evaluated first to ensure that references to the FOR loop variable do not cause side effects.

Usage Notes

- A beginning label used with the LEAVE statement lets you perform a controlled exit from a FOR loop.
- The FOR loop variable-name must exist as a declared updatable local (or global) variable.
- The FOR loop variable can be declared as a numeric value (TINYINT, SMALLINT, INTEGER, BIGINT, FLOAT, REAL, DOUBLE, NUMERIC, NUMBER, or DECIMAL) with no fractional portion.

The following INTERVAL data types are also legal for this type of FOR loop

- INTERVAL YEAR
- INTERVAL MONTH
- INTERVAL DAY
- INTERVAL HOUR
- INTERVAL MINUTE
- INTERVAL SECOND

If INTERVAL is used then the initial and final values must be of the same type. That is, the expressions must have the same data type as the loop variable.

- Within the body of the FOR loop, the FOR loop variable-name cannot be updated using any of the following:
 - The SET statement
 - The GET DIAGNOSTICS statement
 - The INTO clause of the INSERT RETURNING, UPDATE RETURNING, or SELECT statements

In addition, the FOR loop variable name cannot be changed if it is the target of an INOUT or OUT parameter of the CALL statement.

FOR (Counted) Control Statement

In other words, the FOR loop variable behaves like a constant variable within the loop. However, outside the loop, the variable can be modified because the read-only nature of the loop variable is temporary.

- The loop body will not execute if any one of the following is true:
 - The starting value expression evaluates to NULL.
 - The ending value expression evaluates to NULL.
 - The starting value expression is greater than the ending value expression in a forward loop (one that does not contain the REVERSE keyword).
 - When the loop variable is numeric, the value expressions can be any compatible numeric data type. For instance floating point or scaled numeric values can be used.
 - The starting value expression is less than the ending value expression in a reverse loop (one that contains the REVERSE keyword).
- The FOR loop uses the keyword TO as a separator between the initial and final value expressions. This same keyword is used to separate the field names in an interval qualifier. Therefore, there is an ambiguity possible when an apparently well formed expression is used.

```
SQL> begin
cont> declare :i interval year;
cont> for :i in interval'1' year to interval'4'year
for :i in interval'1' year to interval'4'year
      ^
%SQL-W-LOOK_FOR_STT, Syntax error, looking for:
%SQL-W-LOOK_FOR_CON,          MONTH,
%SQL-F-LOOK_FOR_FIN,    found INTERVAL instead
```

This occurs because the TO separator is interpreted as part of the INTERVAL literal or expression. Programmers must enclose the initial expression in parentheses to avoid this ambiguity if it ends with an interval qualifier.

- The STEP value expression is evaluated before the loop variable is assigned a value. The value must be greater than zero and never NULL. If these constraints are violated a runtime error is reported as shown in this simple example.

FOR (Counted) Control Statement

```
SQL> begin
cont> declare :l, :s integer;
cont>
cont> -- set the step size
cont> set :s = 0;
cont>
cont> for :l in reverse 1 to 10 step :s
cont> do
cont>     trace :l;
cont> end for;
cont> end;
%RDB-E-NOT_VALID, validation on field STEP caused operation to fail
SQL>
```

Examples

Example 1: Using a Reverse Loop

```
SQL> SET FLAGS 'TRACE';
SQL> BEGIN
cont> DECLARE :LOOP_VAR INTEGER;
cont> FOR :LOOP_VAR IN REVERSE 1 TO 5
cont> DO
cont>     TRACE :LOOP_VAR;
cont> END FOR;
cont> END;
~Xt: 5
~Xt: 4
~Xt: 3
~Xt: 2
~Xt: 1
```

Example 2: Using an INTERVAL type as the loop variable

```
SQL> begin
cont> declare :i interval year;
cont> for :i in (interval'1' year) to (interval'4'year)
cont> do
cont>     trace :i;
cont> end for;
cont> end;
~Xt: 01
~Xt: 02
~Xt: 03
~Xt: 04
```

FOR (Counted) Control Statement

Example 3: Using a complex expression as the STEP expression

```
SQL> begin
cont> declare :i interval year;
cont> declare :k interval year = interval'18'year;
cont> declare :j integer = 2;
cont>
cont> for :i in (interval'1' year) to :k/2 step :j*2
cont> do
cont>     trace :i;
cont> end for;
cont> end;
~Xt: 01
~Xt: 05
~Xt: 09
```

GET DIAGNOSTICS Statement

Extracts diagnostic information about the execution of the previous SQL statement or SQL routine environment.

The GET DIAGNOSTICS statement captures diagnostic information from an Oracle Rdb maintained data structure called the **diagnostics area**. In the ANSI/ISO SQL standard, the diagnostics area consists of two components: a single header area and an array of detail areas. Oracle Rdb extracts information only from the header component and the first element of the detail area (Exception 1):

- Header area
 - Contains status information about rows and transactions, for example, the number of rows affected by an INSERT, UPDATE, or DELETE statement or the type of transaction that is active.
 - See the statement-item-name argument for a complete list of the status information you can retrieve from the header area.
- Detail area (Exception 1)
 - Contains diagnostic information that corresponds to the status that would be reported in the SQLSTATE or SQLCODE status parameter. The EXCEPTION . . . RETURNED_SQLSTATE argument retrieves the SQLSTATE status information from the detail area. The EXCEPTION . . . RETURNED_SQLCODE argument retrieves the SQLCODE status information from the detail area.

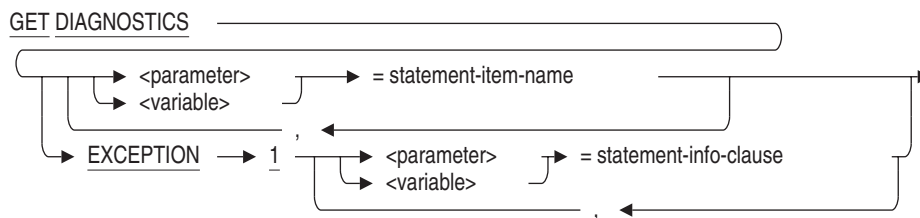
Environment

You can use the GET DIAGNOSTICS statement only within the compound statement of a multistatement procedure:

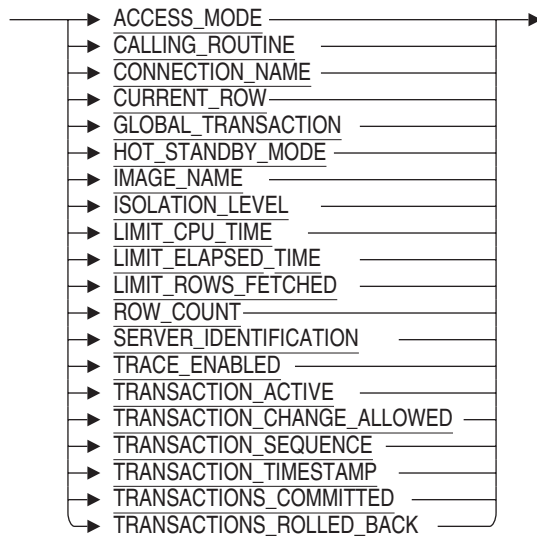
- In interactive SQL
- Embedded in host language programs to be precompiled
- As part of a multistatement procedure in an SQL module
- In dynamic SQL as a statement to be dynamically executed

GET DIAGNOSTICS Statement

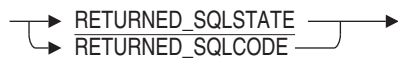
Format



statement-item-name =



statement-info-clause =



GET DIAGNOSTICS Statement

Arguments

IMAGE_NAME

Requests that the activating image name be returned to the caller. The image name includes the node name from which the attach was started. This might be a node different to that on which the Oracle Rdb server is running.

The data is returned to the caller as a VARCHAR (255) value and should be assigned to either a VARCHAR or CHAR data type that supports the ASCII character set.

EXCEPTION

Returns the exception condition following the execution of an SQL statement (other than GET DIAGNOSTICS). Either SQLCODE or SQLSTATE can be returned.

parameter = statement-item-name

variable = statement-item-name

Retrieves information about the statement execution recorded in the diagnostics area and stores it in a simple target specification (a parameter or variable).

RETURNED_SQLCODE

Requests the SQLCODE be returned to the target variable or parameter. The data type of the returned information is INTEGER. Oracle Rdb only returns success (0) and warning status (positive value) for SQLCODE. Any error status will cause the compound statement, or stored routine to return to the calling application.

RETURNED_SQLSTATE

Requests the SQLSTATE be returned to the target variable or parameter. The data type of the returned information is CHAR(5). Oracle Rdb only returns success ('00000') and warning status for SQLSTATE. Any error status will cause the compound statement, or stored routine to return to the calling application.

statement-item-name

Specifies the kind of diagnostic information you can retrieve about a previously executed SQL statement. You can gather the following diagnostic data:

- **ACCESS_MODE** returns the character string READ ONLY, READ WRITE, or BATCH UPDATE to indicate the type of transaction that is active. These character strings are of the CHAR data type. The argument also returns NONE when no transaction is active. See the SET TRANSACTION Statement for a description of transaction access modes.

GET DIAGNOSTICS Statement

- `CALLING_ROUTINE` returns a string of data type `CHAR(31)` of the name of the calling routine. If there is no name for the calling routine, spaces are returned.
- `CONNECTION_NAME` returns the current connection name.
- `CURRENT_ROW` returns the integer value for the number of rows that have been fetched by the innermost `FOR` control statement.
- `GLOBAL_TRANSACTION` returns an integer of 1 when a global transaction is active and an integer of 0 otherwise.
- `ISOLATION_LEVEL` returns the character string `READ COMMITTED`, `REPEATABLE READ`, or `SERIALIZABLE` to indicate the isolation level of a transaction. These character strings are of the `CHAR` data type. The argument also returns `NONE` when no transaction is active. See the `SET TRANSACTION` Statement for a description of transaction isolation levels.
- `LIMIT_CPU_TIME` returns an `INTEGER` value for the session's execution CPU time limit in seconds. If zero (0) is returned it is equivalent to no CPU time limit. This value is established by either the logical name `RDMS$BIND_QG_EXEC_CPU_TIMEOUT` or the `SET QUERY EXECUTION LIMIT CPU TIME` statement.
- `LIMIT_ELAPSED_TIME` returns an `INTEGER` value for the session's execution elapsed time limit in seconds. If zero (0) is returned it is equivalent to no elapsed time limit. This value is established by either the logical name `RDMS$BIND_QG_EXEC_ELAPSED_TIMEOUT` or the `SET QUERY EXECUTION LIMIT ELAPSED TIME` statement.
- `LIMIT_ROWS_FETCHED` returns a `BIGINT` value for the session's row limit. If zero (0) is returned, it is equivalent to no row limit. This value is established by the logical name `RDMS$BIND_QG_REC_LIMIT`.
- `ROW_COUNT` returns an integer for the number of rows affected by an `INSERT`, searched `UPDATE`, searched `DELETE`, or a `FOR` cursor loop statement.
- `TRACE_ENABLED` returns an `INTEGER` value to indicate if the `TRACE` flag has been enabled using the statement `SET FLAGS 'TRACE'`, or by either of the logical names `RDMS$SET_FLAGS` or `RDMS$DEBUG_FLAGS`. A zero (0) is returned if the flag is disabled, otherwise a one (1) is returned to indicate that tracing is enabled.
- `TRANSACTION_ACTIVE` returns an integer of 1 when a transaction is active and an integer of 0 otherwise.

GET DIAGNOSTICS Statement

- **TRANSACTIONS_COMMITTED** returns an integer value for the number of transactions that have been committed during the processing of a multistatement procedure.
- **TRANSACTIONS_ROLLED_BACK** returns an integer value for the number of transactions that have been rolled back during the processing of a multistatement procedure.
- **TRANSACTION_CHANGE_ALLOWED**

There are many situations where the SQL language programmer would like to start or end a transaction but does not know if a transaction statement (**SET TRANSACTION**, **START TRANSACTION**, **COMMIT** or **ROLLBACK**) is currently permitted. The transaction statements are not permitted in the following cases:

 - During a multi-database or global transaction. In this case the transaction must be coordinated by the client, not a server-based procedure.
 - When a **BEGIN ATOMIC** compound statement is in the outer scope.
 - When a **FOR** cursor loop is active in an outer scope.

The **TRANSACTION_CHANGE_ALLOWED** clause allows the programmer to detect these restricted locations and conditionally execute a **COMMIT**, **ROLLBACK**, **START TRANSACTION** or **SET TRANSACTION** as needed.

The result data type is **INTEGER**. If transaction changes are permitted then a value one (1) will be assigned. Otherwise the result will be zero (0).
- **HOT_STANDBY_MODE**

This option returns a text string that indicates if this database is participating in a Hot Standby configuration as master (returns 'MASTER'), or as standby (returns 'STANDBY'), or is not in such a configuration (returns 'NONE').

The result data type is **CHAR** (31).
- **SERVER_IDENTIFICATION**

This option returns a text string containing the Oracle Rdb release number. This is useful for log file annotation.

The result data type is **CHAR** (31).
- **TRANSACTION_TIMESTAMP**

GET DIAGNOSTICS Statement

This option returns the date and time that the last transaction was started. If a transaction is not active, the returned date and time may be for a prior transaction.

Note

The database server will start transactions when performing database operations. So, this timestamp may reflect the time of an internal transaction.

If the default date format is SQL99, this option returns a value with the data type `TIMESTAMP(2)`, otherwise it returns a `DATE (VMS)` data type. The default date format can be changed using either the `SET DIALECT` or `SET DEFAULT DATE FORMAT` statements, or one of the associated module attributes.

- **TRANSACTION_SEQUENCE**

This is the transaction sequence number (TSN) assigned to the most recently started transaction. The TSN is a unique indicator of database transaction activity, however, please note that the TSN may be reused in some cases. The TSN for a `READ ONLY` transaction reflects the transaction state which is visible to the transaction and, therefore, it was previously assigned to a `READ WRITE` transaction. If a `READ WRITE` transaction does not perform database I/O or was rolled back, that TSN may be reused by a subsequent `READ WRITE` transaction.

This option returns a `BIGINT` data type.

Usage Notes

- The diagnostics area is cleared at the beginning of each multistatement procedure.
- You can use the `GET DIAGNOSTICS` statement only within the compound statement of a multistatement procedure.
- Because an exception causes a multistatement procedure to terminate immediately, `RETURNED_SQLCODE` or `RETURNED_SQLSTATE` only returns a warning message. If the procedure is successful, `RETURNED_SQLCODE` or `RETURNED_SQLSTATE` returns a success message.

GET DIAGNOSTICS Statement

Examples

Example 1: Using a GET DIAGNOSTICS statement to retrieve row count

```
PROCEDURE increate_nh (SQLSTATE, :rows_affected INTEGER);
BEGIN ATOMIC
    UPDATE salary_history
    SET     salary_amount = salary_amount * 1.05
    WHERE  salary_end IS NULL
        AND employee_id IN (SELECT  employee_id
                            FROM      employees
                            WHERE     state = 'NH' );
    GET DIAGNOSTICS :rows_affected = ROW_COUNT;
END;
```

Example 2: Using RETURNED_SQLSTATE

```
SQL> DECLARE :Y CHAR(5);
SQL> BEGIN
cont> SET :Y = 'Hello';
cont> GET DIAGNOSTICS EXCEPTION 1 :Y = RETURNED_SQLSTATE;
cont> END;
SQL> PRINT :Y;
    Y
    00000
SQL>
```

Example 3: Using RETURNED_SQLCODE

```
SQL> DECLARE :X INTEGER;
SQL> BEGIN
cont> SET :X = 100;
cont> GET DIAGNOSTICS EXCEPTION 1 :X = RETURNED_SQLCODE;
cont> END;
SQL> PRINT :X;
    X
    0
```

Example 4: Returning the current connection name

```
SQL> CONNECT TO 'ATTACH FILENAME mf_personnel' AS 'my_connection';
SQL> DECLARE :conn_name VARCHAR(20);
SQL> BEGIN
cont>     GET DIAGNOSTICS :conn_name = CONNECTION_NAME;
cont> END;
SQL> PRINT :conn_name;
    CONN_NAME
    my_connection
```

GET DIAGNOSTICS Statement

Example 5: Using the TRANSACTION_TIMESTAMP and TRANSACTION_SEQUENCE options

```
SQL> set transaction read write;
SQL> show transaction
Transaction information:
    Statement constraint evaluation is off
On the default alias
Transaction characteristics:
    Read Write
Transaction information returned by base system:
a read-write transaction is in progress
- updates have not been performed
- transaction sequence number (TSN) is 0:256
- snapshot space for TSNs less than 0:256 can be reclaimed
- recovery unit journal filename is USER2:[RDM$RUJ]SCRATCH$00018679B3AD.RUJ;1
- session ID number is 8
SQL>
SQL> declare :x date vms;
SQL>
SQL> begin get diagnostics :x = transaction_timestamp; end;
SQL> print :x;
    X
    27-MAY-1999 22:39:17.02
SQL>
SQL> declare :y bigint;
SQL>
SQL> begin get diagnostics :y = transaction_sequence; end;
SQL> print :y;
                Y
                256
SQL>
SQL> select current_timestamp from rdb$database;
    27-MAY-1999 22:39:18.20
1 row selected
SQL>
SQL> commit;
```

Example 6: Using the HOT_STANDBY_MODE and SERVER_IDENTIFICATION options

```
SQL> set flags 'trace';
SQL> declare :id, :hsmode char(31);
SQL> begin
cont> get diagnostics :id = SERVER_IDENTIFICATION,
cont>                  :hsmode = HOT_STANDBY_MODE;
cont> trace :id, :hsmode;
cont> end;
~Xt: Oracle Rdb V7.1                NONE
```


GET DIAGNOSTICS Statement

Example 7: Using the LIMIT_CPU_TIME, LIMIT_ROWS_FETCHED, and LIMIT_ELAPSED_TIME options

```
SQL> set flags 'trace';
SQL> set query execution limit elapsed time 10 minutes;
SQL> begin
cont> declare :row_limit integer;
cont> declare :elapsed_limit integer;
cont> declare :cpu_limit integer;
cont> get diagnostics
cont>      :cpu_limit = LIMIT_CPU_TIME,
cont>      :row_limit = LIMIT_ROWS_FETCHED,
cont>      :elapsed_limit = LIMIT_ELAPSED_TIME;
cont> trace 'LIMIT_ROWS_FETCHED: ', :row_limit;
cont> trace 'LIMIT_CPU_TIME:      ', :cpu_limit;
cont> trace 'LIMIT_ELAPSED_TIME: ', :elapsed_limit;
cont> end;
~Xt: LIMIT_ROWS_FETCHED: 0
~Xt: LIMIT_CPU_TIME:      0
~Xt: LIMIT_ELAPSED_TIME: 600
SQL>
```

Example 8: Using the TRACE_ENABLED keyword in a compound statement

```
SQL> declare :x integer;
SQL> begin
cont> get diagnostics :x = TRACE_ENABLED;
cont> end;
SQL> print :x;
      X
      0

SQL> set flags 'trace';
SQL> begin
cont> get diagnostics :x = TRACE_ENABLED;
cont> end;
SQL> print :x;
      X
      1
```

GET ENVIRONMENT Statement

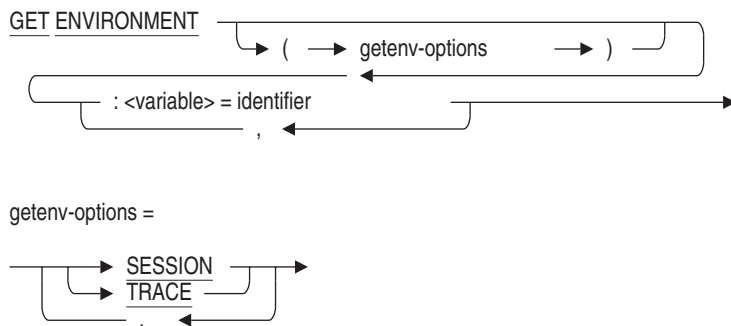
GET ENVIRONMENT Statement

Loads values defined by OpenVMS DCL symbols or logical names and SQL session values into locally declared SQL variables.

Environment

You can use the GET ENVIRONMENT statement in interactive SQL only.

Format



Arguments

SESSION

Directs GET ENVIRONMENT to return selected interactive SQL session options. These options can be used to save and restore session state during the execution of an SQL script.

TRACE

Displays the translated string value prior to being converted to the data type of the variable. This can assist in diagnosing data conversion errors. The display will indicate if the result was derived from a local symbol, global symbol, logical name, or session value. For example:

```
SQL> GET ENVIRONMENT (TRACE)
cont>      :xx indicator :xx_ind = XX;
01: XX = XX "--" (Local)
%RDB-E-ARITH_EXCEPT, truncation of a numeric value at runtime
-COSI-F-INPCONERR, input conversion error
```

GET ENVIRONMENT Statement

Usage Notes

The following table shows the associated SET command which will accept the output from GET ENVIRONMENT (SESSION). These commands allow application to re-establish the environment after using SET commands within a SQL script. Refer to the listed SET command for details of the string value that will be returned from GET ENVIRONMENT.

Table 7–4 GET ENVIRONMENT session keywords

SESSION Keyword	Associated SET command
DEFAULT_CATALOG	SET CATALOG
CONSTRAINT_MODE	SET DEFAULT CONSTRAINT MODE
CHARACTER_LENGTH	SET CHARACTER LENGTH
COMPOUND_TRANSACTIONS	SET COMPOUND TRANSACTION
DATE_FORMAT	SET DEFAULT DATE FORMAT
DEFAULT_CONSTRAINT_MODE	SET DEFAULT CONSTRAINT MODE
DIALECT	SET DIALECT
HOLD_CURSORS	SET HOLD CURSOR
NULL_STRING	SET DISPLAY NULL STRING
QUIET_COMMIT	SET QUIET COMMIT
QUOTING_RULES	SET QUOTING RULES
KEYWORD_RULES	SET KEYWORD RULES
DEFAULT_SCHEMA	SET SCHEMA
DEFAULT_ALIAS	SET ALIAS

- If no NULL indicator is specified and the DCL symbol or logical name is not found, an error will be reported. For example:

```
SQL> GET ENVIRONMENT
cont> :x = THE_TIME;
%SQL-F-UNDEFVAR, Variable THE_TIME is not defined
-LIB-F-NOSUCHSYM, no such symbol
```

- If the specified symbol is not defined, and an INDICATOR is specified for the variable, the indicator will be set, but the variable will remain unchanged. For example:

GET ENVIRONMENT Statement

```
SQL> GET ENVIRONMENT :xx indicator :xx_ind = XX;
SQL>
SQL> PRINT :xx, :xx_ind;
      XX      XX_IND
      0      1
```

If the TRACE option is used, the value will be displayed as NULL for the symbol.

- The specified variable must be a local variable defined using the DECLARE statement. For example:

```
SQL> DECLARE :xx, :xx_ind INTEGER;
```

The identifier is assumed to be a DCL symbol or logical name. It is first translated as a symbol name and, if that fails, it is translated as a logical name. If translation is successful, the string result is converted to the data type of the variable. The name must conform to the rules defined by the OpenVMS DCL naming conventions.

Multiple assignments can be specified, separated by commas.

Examples

Example 1: Using the GET ENVIRONMENT Statement

```
$ emp_id = "00164"
$ SQL$
SQL> ATTACH 'FILENAME MF_PERSONNEL';
SQL> DECLARE :e CHAR(5);
SQL> GET ENVIRONMENT :e = emp_id;
SQL> SELECT last_name, first_name FROM employees WHERE employee_id = :e;
  LAST_NAME      FIRST_NAME
  Toliver        Alvin
1 row selected
SQL> ROLLBACK;
```

Example 2: Using the SESSION option

This example uses the SESSION option to save the DIALECT and restore it upon completion of the SQL script.

```
SQL> declare :Rdb_DIALECT char(10);
SQL> get environment (session) :Rdb_DIALECT = DIALECT;
SQL> set dialect 'SQL92';
SQL> -- get SQL92 semantics for UNIQUE constrain
SQL> create table T (a integer unique);
SQL> set dialect :Rdb_DIALECT;
SQL> undeclare :Rdb_DIALECT;
```

GRANT Statements

Adds privileges or roles to object access control.

Usage Notes

The following notes apply to all GRANT statements.

- You cannot execute the GRANT statement when any of the LIST, DEFAULT or RDB\$SYSTEM storage areas are set to read-only. You must first set these storage areas to read/write. Note that in some databases RDB\$SYSTEM will also be the default and list storage area.
- Users with the OpenVMS SYSPRV privilege implicitly receive the same privileges as users with the DBADM database privilege.
Users with the OpenVMS OPER privilege implicitly receive the SELECT, INSERT, UPDATE and DELETE database privileges.
Users with the OpenVMS SECURITY privilege implicitly receive the same privileges as users with the SECURITY database privilege.
Users with the OpenVMS BYPASS privilege implicitly receive *all* privileges except the Oracle Rdb DBADM and SECURITY database privileges and the DBCtrl database and table privileges.
Users with the OpenVMS READALL privilege implicitly receive Oracle Rdb SELECT and SHOW database and table privileges.
- For the SELECT, INSERT, UPDATE and DELETE data manipulation privileges, SQL checks the access privilege set for the database and for the individual table before allowing access to a specific table. For example, if your SELECT privilege for a database that contains the EMPLOYEES table is revoked, you will not be able to read rows from the table even though you may have SELECT privilege to the EMPLOYEES table itself.
- Additions and changes to ACLs do not take effect until you attach to the database again, even though those changes are displayed by the SHOW PROTECTION and SHOW PRIVILEGES statements. Additions and changes to ACLs do not take effect for other users until they attach to the database again.

GRANT Statements

- You must execute the GRANT statement in a read/write transaction. If you issue this statement when there is no active transaction, SQL starts a transaction with characteristics specified in the most recent DECLARE TRANSACTION statement.

GRANT Statement

Creates or adds privileges to an entry to the Oracle Rdb access privilege set, called the **access control list (ACL)**, for a database, table, view, column, module, or external routine. Each entry in an ACL consists of an identifier and a list of privileges assigned to the identifier:

- Each identifier specifies a user or a set of users.
- The list of privileges specifies which operations that user or user group can perform on the database, table, view, column, module, or external routine.

When a user tries to perform an operation on a database, SQL reads the associated ACL from top to bottom, comparing the identifier of the user with each entry. As soon as SQL finds the first match, it grants the rights listed in that entry and stops the search. All identifiers that do not match a previous entry “fall through” to the entry `[*,*]` (equivalent to the SQL keyword PUBLIC). If no entry has the identifier `[*,*]`, then users with unmatched identifiers are denied all access to the database, table, view, column, module, or external routine.

For this reason, both the entries and their order in the list are important.

Under the Oracle Rdb default protection scheme, when you create a new database, table, view, module, or external routine, you get all access rights to that object, including DBCTRL. All other users of that object are given no access rights to it. For any tables or views created under the Oracle Rdb default protection scheme, the creator of the table or view receives all the access rights to the object, including DBCTRL, and all other users receive no access rights to the object.

The DBCTRL access right enables an object’s creator to grant DBCTRL to other users. See the Usage Notes for information on how you can tailor the default protection for any new tables that you create within a database.

To remove privileges from or entirely delete an entry to the Oracle Rdb access privilege set for a database, table, column, module, or external routine, see the REVOKE Statement.

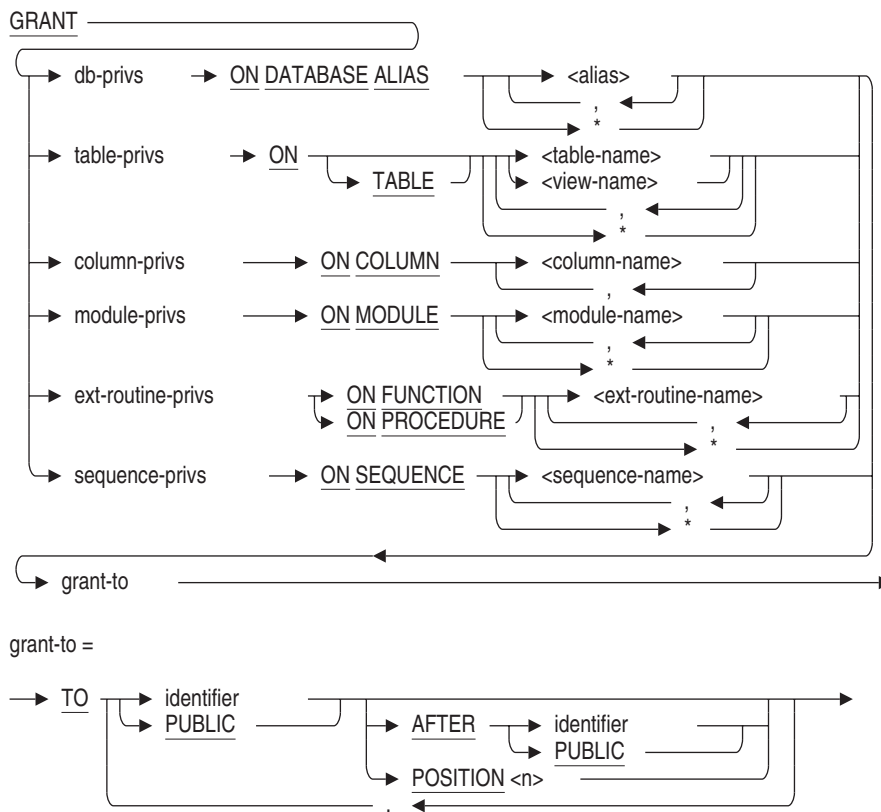
GRANT Statement

Environment

You can use the GRANT statement:

- In interactive SQL
- Embedded in host language programs to be precompiled
- As part of a nonstored procedure in a nonstored SQL module
- In dynamic SQL as a statement to be dynamically executed

Format



GRANT Statement

db-privs =

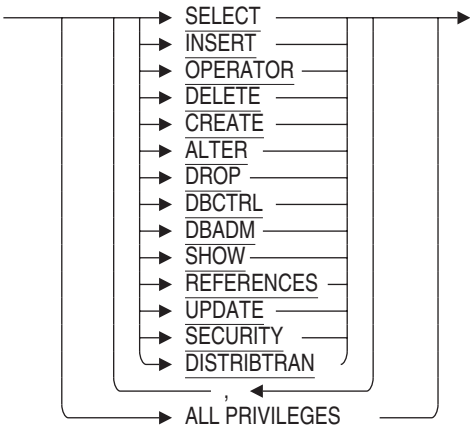
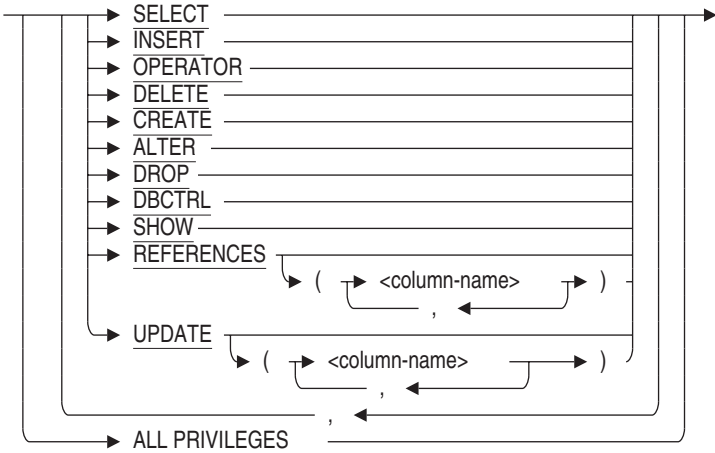
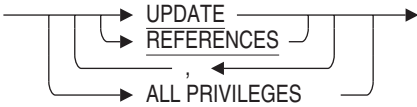


table-privs=



column-privs=



GRANT Statement

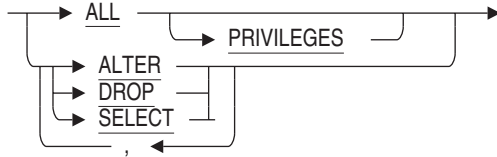
module-privs =



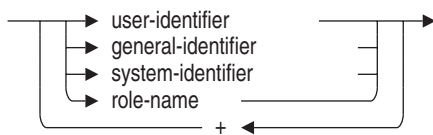
ext-routine-privs =



sequence-privs =



identifier =



Arguments

AFTER identifier

AFTER PUBLIC

POSITION n

Specifies the position of the entry within the ACL to be modified or created.

GRANT Statement

With the **AFTER** or **POSITION** argument, you can specify the position in the list after which SQL searches for an ACL entry with an identifier that matches the one specified in the **TO** clause of the **GRANT** statement.

Following are specifics about the **AFTER** and **POSITION** arguments:

- In the **AFTER** argument, the identifier specifies the entry in the ACL after which SQL begins its search for the entry to be modified or created. If none of the entries in the ACL has an identifier that matches the identifier specified in the **AFTER** argument, SQL generates an error and the statement fails.

Starting *after* the entry specified by the identifier in the **AFTER** argument, SQL searches the entries in the ACL. If an entry has an identifier that matches the identifier specified by the **TO** clause of the **GRANT** statement, SQL creates a new entry that contains only those privileges specified in the **GRANT** statement. SQL retains only the entry appearing first in the ACL, and deletes any entries with duplicate identifiers.

If none of the entries has an identifier that matches the identifier specified by the **TO** clause of the **GRANT** statement, SQL creates a new ACL entry immediately following the identifier specified in the **AFTER** argument.

Specifying **PUBLIC** is equivalent to a wildcard specification of all user identifiers.

- In the **POSITION** argument, the integer specifies the earliest relative position in the ACL of the entry to be modified or created.

Starting *with* the position specified by the **POSITION** argument, SQL searches the entries in the ACL. If an entry has an identifier that matches the identifier specified by the **TO** clause of the **GRANT** statement, SQL creates a new entry that contains only those privileges specified in the **GRANT** statement. SQL retains only the entry appearing first in the ACL, and deletes any entries with duplicate identifiers.

If none of the entries has an identifier that matches the identifier specified by the **TO** clause of the **GRANT** statement, SQL creates a new entry for that identifier at the relative position specified in the **POSITION** argument (even if an entry before the position at which SQL began its search had an identifier that matched).

If you specify a position higher than the number of entries in the list, SQL places the entry last in the ACL. For example, if you specify position 12 and there are only 10 entries in the list, the new entry is placed in position 11 and given that position number.

GRANT Statement

- If you omit the AFTER or POSITION argument, SQL searches the entire ACL for an identifier list that matches the one specified in the TO clause of the GRANT statement. If it finds a match, it modifies the ACL entry by adding those privileges specified in the privilege list that are not already present. If there is no match, SQL creates a new entry at the beginning of the ACL.

ALL PRIVILEGES

Specifies that SQL should grant all privileges in the ACL entry.

general-identifier

Identifies groups of users on the system and are defined by the OpenVMS system manager in the system rights database. The following are possible general identifiers:

DATAENTRY
SECRETARIES
MANAGERS

ON SEQUENCE sequence-name

Specifies whether the GRANT statement applies to ACLs for the named sequence or sequences.

ON DATABASE ALIAS *

ON TABLE *

ON MODULE *

ON FUNCTION *

ON PROCEDURE *

ON SEQUENCE *

Specifies whether the GRANT statement applies to ACLs for all objects of the specified type. If privileges are denied for the operation on some objects, then the GRANT is aborted.

db-privs

table-privs

column-privs

module-privs

ext-routine-privs

sequence-privs

Specifies the list of privileges you want to add to an existing ACL entry or create in a new one. The operations permitted by a given privilege keyword differ, depending on whether you granted it for a database, table, column, module, external routine, or sequence.

GRANT Statement

Table 7–5 lists the privilege keywords and their meanings for databases, tables, columns, modules, external routines, and sequences.

Table 7–5 SQL Privileges for Databases, Tables, Columns, Modules, External Routines and Sequences

Privilege	For the Access Privilege Set of a Database, Grants the Privilege to:	For the Access Privilege Set of a Table, Column, View, Module, External Routine or Sequence, Grants the Privilege to:
ALTER	Change database parameters or change a domain.	Alter the table, index, or storage map. Alter a module, external routine, or sequence. Does not apply to column privileges.
CREATE	Create a catalog, schema, table, domain, collating sequence, storage area, external routine, module, or sequence.	Create a view, trigger, index, sequence, storage map, or outline that uses a table. Does not apply to column privileges.
DBADM	Perform any data manipulation or data definition operation on any named object. Override many database privileges.	Not applicable, but syntactically allowed.
DBCTRL	Create, delete, or modify an access privilege set entry for the database.	Grant or revoke an access privilege set entry for the table, sequence, module, or external routine. Does not apply to column privileges.
DELETE	Delete data from a table defined in the database.	Delete data from a table. Does not apply to column privileges.
DISTRIBTRAN	Run a distributed (two-phase commit protocol) transaction against the database.	Not applicable.

(continued on next page)

GRANT Statement

Table 7–5 (Cont.) SQL Privileges for Databases, Tables, Columns, Modules, External Routines and Sequences

Privilege	For the Access Privilege Set of a Database, Grants the Privilege to:	For the Access Privilege Set of a Table, Column, View, Module, External Routine or Sequence, Grants the Privilege to:
DROP	Delete a catalog, schema, domain, collating sequence, or path name.	Delete the table, index or outline that uses a table. Delete a view, column, constraint, trigger, sequence, or storage map. Delete a view, module, external routine, or sequence.
EXECUTE	Not applicable.	Allow the execution of a module or external routine. Does not apply to column, sequence, or table privileges.
INSERT	Store data in a table defined in the database.	Store data in the table. Does not apply to column or sequence privileges.
REFERENCES	Not applicable, but syntactically allowed.	Define constraints that refer to data in a table or column. Define tables using the LIKE clause. Define synonyms that reference those objects.
SECURITY	Override many database privileges.	Not applicable.
SELECT	Attach to a database and read data from a table defined in the database.	Read data from a table or reference the NEXTVAL and CURRVAL pseudocolumns in a sequence. Does not apply to column privileges.

(continued on next page)

GRANT Statement

Table 7–5 (Cont.) SQL Privileges for Databases, Tables, Columns, Modules, External Routines and Sequences

Privilege	For the Access Privilege Set of a Database, Grants the Privilege to:	For the Access Privilege Set of a Table, Column, View, Module, External Routine or Sequence, Grants the Privilege to:
SHOW	Not applicable. Syntactically allowed, but not implemented. Reserved for future versions.	Not applicable. Syntactically allowed, but not implemented. Reserved for future versions.
UPDATE	Update data in a table defined in the database.	Update data in a table or column.

Privileges on a column are determined by the privileges defined for the table combined with those specified for the specific column ACL.

The SELECT privilege is a prerequisite for all other data manipulation privileges, except UPDATE and REFERENCES. If you do not grant the SELECT privilege, you effectively deny SELECT, INSERT, and DELETE privileges, even if they are specified in the privilege list. It is not possible for you to deny yourself the SELECT privilege.

For the SELECT, INSERT, UPDATE, and DELETE data manipulation privileges, SQL checks the ACL for the database and for the individual table before allowing access to a specific table. For example, if you are granted SELECT privilege for the EMPLOYEES table, you are not able to select rows from the table unless you also have SELECT privilege for the database that contains the EMPLOYEES table.

A user with the UPDATE privilege on the table automatically receives the UPDATE privilege on all columns in the table. To update a column, you must have the UPDATE privilege either for the column or the table. However, you can restrict the UPDATE privileges by defining them only on specific columns you want users to be able to update, and by removing the UPDATE privilege from the table entry.

You can modify the data in a column only with the UPDATE privilege on the column and the SELECT privilege on the database.

The REFERENCES privilege lets you define a constraint for a database with ANSI/ISO-style privileges. For a database with ACL-style privileges, you need the CREATE privilege to define a constraint.

GRANT Statement

You cannot deny yourself the DBCTRL privilege for a database or table that you create. This restriction may cause GRANT statements to fail when you might expect them to work.

For instance, suppose an ACL has no entry for PUBLIC. The following GRANT statement fails because it creates an entry for PUBLIC at the top of the ACL that does not include the DBCTRL privilege, effectively denying DBCTRL to all other entries on the list, including the owner:

```
SQL> GRANT SELECT, INSERT ON EMPLOYEES TO PUBLIC;  
%RDB-E-NO_PRIV, privilege denied by database facility
```

role-name

The name of a role, such as one created with the CREATE ROLE statement or one that can be created automatically. (If the role name exists as an operating system group or rights identifier, then Oracle Rdb will create the role automatically when you issue the GRANT statement. A role that is created automatically always has the attribute of IDENTIFIED EXTERNALLY.)

TO identifier

TO PUBLIC

Specifies the identifiers for the new or modified ACL entry. Specifying PUBLIC is equivalent to a wildcard specification of all user identifiers.

You can specify four types of identifiers:

- User identifiers
- General identifiers
- System-defined identifiers
- Role names

You can specify more than one identifier by combining them with plus signs (+). Such identifiers are called multiple identifiers. They identify only those users who are common to all the groups defined by the individual identifiers. Users who do not match all the identifiers are not controlled by that entry.

For instance, the multiple identifier SECRETARIES + INTERACTIVE specifies only members of the group defined by the general identifier SECRETARIES that are interactive processes. It does not identify members of the SECRETARIES group that are not interactive processes.

The following arguments briefly describe the three types of identifiers. For more information about identifiers, see your operating system documentation.

GRANT Statement

system-identifier

System-defined identifiers are automatically defined by the system when the rights database is created at system installation time. System-defined identifiers are assigned depending on the type of login you execute. The following are all valid system-defined identifiers:

```
BATCH
NETWORK
INTERACTIVE
LOCAL
DIALUP
REMOTE
```

user-identifier

Identifies each user on the system.

The user identifier consists of the standard OpenVMS user identification code (UIC), a group name and a member name (user name). The group name is optional. The user identifier can be in either numeric or alphanumeric format. The following are all valid user identifiers that could identify the same user:

```
K_JONES
[SYSTEM3, K_JONES]
[341,311]
```

You can use the asterisk (*) wildcard character as part of a user identifier. For example, if you want to specify all users in a group, you can enter [system3, *] as the identifier.

When Oracle Rdb creates a database, it automatically creates an ACL entry with the identifier [*,*] (also known as PUBLIC), which specifies the privileges given to all users on the system.

You cannot use more than one user identifier in a multiple identifier.

Usage Notes

- Additions and changes to ACLs do not take effect until you attach to the database again, even though those changes are displayed by the SHOW PROTECTION and SHOW PRIVILEGES statements. Additions and changes to ACLs do not take effect for other users until they attach to the database again.

GRANT Statement

- You must attach to all databases that you refer to in a GRANT statement. If you use the default alias, you must use the alias RDB\$DBHANDLE to work with database ACLs.
- You can use the GRANT statement to modify existing ACL entries or create new ones.

To modify an existing ACL entry, specify the same identifier in the TO clause as is in the existing entry.

To create a new ACL entry, specify an identifier that is not already part of an entry.

- When you create new tables and views, they have a PUBLIC access of NONE by default.
- To override PUBLIC access for newly created tables, define an identifier with the name DEFAULT in the system privileges database. The access privileges given to this identifier on your database are then assigned to PUBLIC for any newly created tables and views.

You might want to assign the SELECT and UPDATE privileges to the database with alias TEST1. For example:

```
SQL> ATTACH 'ALIAS TEST1 FILENAME personnel';
SQL> SHOW PROTECTION ON DATABASE TEST1.
Protection on Alias TEST1
  (IDENTIFIER=[dbs,smallwood],ACCESS=SELECT+INSERT+UPDATE+DELETE+SHOW+
  CREATE+ALTER+DROP+DBCTRL+OPERATOR+DBADM+REFERENCES+SECURITY)
  (IDENTIFIER=[*,*],ACCESS=NONE)
SQL> GRANT SELECT, UPDATE ON DATABASE ALIAS TEST1
cont> TO DEFAULT;
```

You must commit and disconnect the transaction to make the change in protection occur.

```
SQL> COMMIT;
SQL> DISCONNECT DEFAULT;
```

The protection on existing tables in the database is not changed, but any new tables that you define receive the protection specified by the DEFAULT identifier. In this example, the owner (SMALLWOOD) receives all the access privileges to the new table TABLE1, and all other users receive the SELECT and UPDATE access privileges specified by the DEFAULT identifier.

GRANT Statement

```
SQL> ATTACH 'ALIAS TEST1 FILENAME personnel';
SQL> SET TRANSACTION READ WRITE;
SQL> CREATE TABLE TEST1.TABLE1
cont>      (LAST_NAME_DOM CHAR(5),
cont>      YEAR_DOM SMALLINT);
SQL> SHOW PROTECTION ON TEST1.TABLE1;
Protection on Table TABLE1
      (IDENTIFIER=[dbs,smallwood],ACCESS=SELECT+INSERT+UPDATE+DELETE+SHOW+
      CREATE+ALTER+DROP+DBCTRL+DBADM+REFERENCES+SECURITY)
      (IDENTIFIER=[*,*],ACCESS=SELECT+UPDATE)
```

The DEFAULT identifier is typically present on an OpenVMS system because the DEFAULT account is always present and cannot be removed. However, it is possible to remove the DEFAULT identifier associated with that account. If the DEFAULT identifier was removed from your system, Oracle Rdb returns an error message.

```
SQL> GRANT INSERT ON DATABASE ALIAS TEST1 to DEFAULT;
%SYSTEM-F-NOSUCHID, unknown rights identifier
```

- The DBADM and SECURITY privileges are the two Oracle Rdb role-oriented privileges. Users with these privileges can override ACLs for some objects to perform certain system-level operations. Oracle Rdb role-oriented privileges are limited to the database in which they are granted.

The two role-oriented database privileges cannot override each other. (For example, DBADM privilege does not override SECURITY privilege.)

A user having one of these role-oriented privileges may be implicitly granted certain other Oracle Rdb privileges. An implicit privilege is a privilege granted as a result of an override; the user operates as if the user actually holds the privilege, but the privilege is not explicitly granted and stored in the ACL for the object.

Table 7–6 shows which Oracle Rdb privileges can be overridden by the Oracle Rdb DBADM and SECURITY database privileges and the OpenVMS SYSPRV, BYPASS, and READALL privileges. For each table entry, the question is whether users with the Oracle Rdb or OpenVMS privilege specified in the columns at the top of the table implicitly receive the access rights associated with the Oracle Rdb privilege in the first column of the table.

For example, the Y for the first entry in the table shows that the Oracle Rdb DBADM privilege overrides the Oracle Rdb ALTER privilege, and the N in the second entry shows that the Oracle Rdb SECURITY privilege does not override the Oracle Rdb ALTER privilege. An N/A entry in the table indicates that a privilege cannot override itself.

GRANT Statement

Table 7–6 Privilege Override Capability

Privilege	Database Privilege		OpenVMS Privilege		
	DBADM	SECURITY	SYSPRV	BYPASS	READALL
ALTER	Y	N	Y	Y	N
CREATE	Y	N	Y	Y	N
DBADM	N/A	N	Y	N	N
DBCTRL	Y	Y	Y	N	N
DELETE (database)	Y	Y	Y	Y	N
DELETE (table)	Y	N	Y	Y	N
DISTRIBTRAN	Y	N	Y	Y	N
DROP	Y	N	Y	Y	N
EXECUTE	Y	Y	N	N	N
INSERT (database)	Y	Y	Y	Y	N
INSERT (table)	Y	N	Y	Y	N
REFERENCES	Y	N	Y	Y	N
SECURITY	N	N/A	N	N	N
SELECT (database)	Y	Y	Y	Y	Y
SELECT (table)	Y	N	Y	Y	Y
SHOW	Y	N	Y	Y	Y
UPDATE (database)	Y	Y	Y	Y	N
UPDATE (table)	Y	N	Y	Y	N

- Users with the DBADM database privilege can perform any data definition or data manipulation operation on any named object, including the database, regardless of the ACL for the object. The DBADM privilege is the most powerful privilege in Oracle Rdb because it can override most privilege checks performed by Oracle Rdb. Users with the DBADM database privilege implicitly receive *all* privileges for all objects, except the SECURITY database privilege.
- Users with the SECURITY database privilege implicitly receive the Oracle Rdb SELECT, INSERT, UPDATE, and DELETE database privileges and the Oracle Rdb DBCTRL database and table privileges.
- You must execute the GRANT statement in a read/write transaction. If you issue this statement when there is no active transaction, Oracle Rdb starts a read/write transaction implicitly.

GRANT Statement

- You cannot execute the GRANT statement when the RDB\$SYSTEM storage area is set to read-only. You must first set RDB\$SYSTEM to read/write. See the *Oracle Rdb Guide to Database Maintenance* for more information on the RDB\$SYSTEM storage area.
- You can deny users the right to create databases.
 - You can use the RDBVMS\$CREATE_DB logical name along with the RDBVMS\$CREATE_DB rights identifier to deny users the right to create databases. See the *Oracle Rdb Guide to Database Design and Definition* for more information about the RDBVMS\$CREATE_DB logical name.

Caution

When you use the RDBVMS\$CREATE_DB logical name, other installed third-party products will not be able to use Oracle Rdb to create Oracle Rdb databases. Therefore, you must deassign this logical name whenever users of such products need to create an Oracle Rdb database.

- You cannot GRANT privileges on procedures or functions created by using CREATE MODULE. Use GRANT on the module instead.
- When you grant privileges to a user identifier, if security checking is set to internal, then SQL automatically creates that user as a database user. In effect, it is as though you issued a CREATE USER statement for the system user. See Example 7 in the Examples section.
- Prior to Oracle Rdb Release 7, privileges required for data manipulation operations on global and local temporary tables were the same as those required for base tables. For example, to perform an insert into a global temporary table, a user needed SELECT+INSERT privileges at the database level.

This requirement existed because an insert operation into a base table implicitly inserts data into the database. The privilege granted at the database level was used to filter the privileges for the table.

However, unlike base tables, the data in temporary tables is not actually stored in the database, thus temporary tables never update the database.

GRANT Statement

Beginning with Release 7 of Oracle Rdb, only the privileges associated with the temporary table are considered when performing security validation during data manipulation operations. For example, if a user can attach to the database (requires SELECT privilege only) and is granted INSERT privilege to a global or local temporary table, then the user (or an invoker's rights stored routine) can update the temporary table. This change affects the operation of SQL*Net for Rdb, which no longer requires database manipulation privileges (INSERT, UPDATE, DELETE) for processing temporary tables.

This relaxation of the security checking applies only to temporary tables.

For more information on protection for an Oracle Rdb database, see the chapter on defining privileges in the *Oracle Rdb Guide to Database Design and Definition*.

Examples

Example 1: Redefining a database to make ACL changes take effect

This example illustrates that GRANT and REVOKE statements do not take effect until you attach to the database again.

```
SQL> -- Display the ACL for the EMPLOYEES table:
SQL> SHOW PROTECTION ON TABLE EMPLOYEES;
Protection on Table EMPLOYEES
  (IDENTIFIER=[sql,warring],ACCESS=SELECT+INSERT+UPDATE+DELETE+SHOW+CREATE+
    ALTER+DROP+DBCTRL+DBADM+REFERENCES)
  (IDENTIFIER=[*,*],ACCESS=SELECT+INSERT+UPDATE+DELETE+ALTER+DROP)
SQL>
SQL> -- User warring, the owner of the database, denies
SQL> -- herself INSERT access to the EMPLOYEES table:
SQL> REVOKE INSERT ON TABLE EMPLOYEES FROM warring;
SQL> COMMIT;
SQL>
SQL> -- The SHOW PROTECTION statement displays the change
SQL> -- (INSERT is no longer part of the ACL entry
SQL> -- for warring):
SQL> SHOW PROTECTION ON TABLE EMPLOYEES;
Protection on Table EMPLOYEES
  (IDENTIFIER=[sql,warring],ACCESS=SELECT+UPDATE+DELETE+SHOW+CREATE+ALTER+
    DROP+DBCTRL+DBADM+REFERENCES)
  (IDENTIFIER=[*,*],ACCESS=SELECT+INSERT+UPDATE+DELETE+ALTER+DROP)
SQL>
```

GRANT Statement

```
SQL> -- But the change is not yet effective.
SQL> -- User warring can still store rows in the EMPLOYEES table:
SQL> INSERT INTO EMPLOYEES (EMPLOYEE_ID) VALUES ('99999');
1 row inserted
SQL> SELECT EMPLOYEE_ID
cont> FROM EMPLOYEES
cont> WHERE EMPLOYEE_ID = '99999';
EMPLOYEE_ID
99999
1 row selected
SQL> ROLLBACK;
SQL>
SQL> -- To make the ACL change take effect, issue another ATTACH statement
SQL> -- to override the current declaration:
SQL> ATTACH 'FILENAME personnel';
This database context has already been declared.
Would you like to override this declaration (No)? Y
SQL>
SQL> -- Now warring cannot insert new rows into the EMPLOYEES table:
SQL> INSERT INTO EMPLOYEES (EMPLOYEE_ID) VALUES ("99999");
%RDB-E-NO_PRIV, privilege denied by database facility
SQL>
SQL> -- A GRANT statement gives all privileges back to warring:
SQL> GRANT ALL ON TABLE EMPLOYEES TO warring;
SQL> COMMIT;
```

Example 2: Creating an ACL with an SQL command file

The following SQL command file creates an ACL for the default database by specifying the default alias RDB\$DBHANDLE. It uses two general guidelines for ordering ACL entries:

- The less restrictive the user identifier, the lower on the list that ACL should go.
- The more powerful the privilege, the higher on the list that ACL should go.

Because SQL reads the list from top to bottom, you should place entries with more specific identifiers earlier, and those with more general ones later. For example, if you place the entry with the most general user identifier, [*,*], first in the list, all users match it, and Oracle Rdb grants or denies all the access rights specified there to all users.

Similarly, if you place the general entry [admin,*] before the specific entry [admin,ford], SQL matches user [admin,ford] with [admin,*] and denies the access rights INSERT, UPDATE, and DELETE, which user [admin,ford] needs.

GRANT Statement

```
-- Database Administrator -- needs all privileges.
--
GRANT ALL
ON DATABASE ALIAS RDB$DBHANDLE
TO [group2,adams]
POSITION 1;

-- Assistant -- needs to be able to use data definition statements.
--
GRANT SELECT,CREATE,ALTER,DROP
ON DATABASE ALIAS RDB$DBHANDLE
TO [group2,clark]
POSITION 2;

-- Operator -- needs to be able to perform database maintenance tasks.
--
GRANT SELECT, ALTER, DBADM
ON DATABASE ALIAS RDB$DBHANDLE
TO [group2,lawrence]
POSITION 3;

-- Security Administrator -- needs to specify and show security events
-- audited for a database and review the audit trail.
--
GRANT SECURITY
ON DATABASE ALIAS RDB$DBHANDLE
TO [group2,davis]
POSITION 4;

-- Manager -- needs to be able to use all data manipulation statements.
--
GRANT SELECT,INSERT,UPDATE,DELETE
ON DATABASE ALIAS RDB$DBHANDLE
TO [admin,smith]
POSITION 5;

-- Secretary -- needs to be able to read, write, and delete data.
-- No access to data definition or maintenance.
--
GRANT SELECT,INSERT,UPDATE,DELETE
ON DATABASE ALIAS RDB$DBHANDLE
TO [admin,ford]
POSITION 6;

-- Programmers -- need to perform data definition and data manipulation
-- on some tables and constraints to test application programs.
--
GRANT SELECT,INSERT,UPDATE,DELETE,CREATE,ALTER,DROP,REFERENCES
ON DATABASE ALIAS RDB$DBHANDLE
TO PROGRAMMERS
POSITION 7;
```


GRANT Statement

```
-- Clerks -- need to be able only to read data. No access to modify, erase,
-- store, data definition, or maintenance statements.
--
GRANT SELECT
ON DATABASE ALIAS RDB$DBHANDLE
TO [admin,*]
POSITION 8;

-- Deny access to all users not explicitly granted access to the database.
--
REVOKE ALL
ON DATABASE ALIAS RDB$DBHANDLE
FROM PUBLIC
POSITION 9;
```

Example 3: Granting column access and denying table access

You need the REFERENCES privilege to define constraints that affect a particular column. You need the UPDATE privilege to update data in a column. A user with the UPDATE privilege for a table automatically receives the UPDATE privilege for all columns in that table. To update a column, you must have the UPDATE privilege either for the column or for the table. However, a database administrator can restrict UPDATE privileges by defining them only for columns users should be able to update, and then removing the UPDATE privilege from the table entry. Because current salary is sensitive information, you might want to restrict the ability to update this amount.

The following example prevents user [admin,ford] from updating any column in the SALARY_HISTORY table except SALARY_START and SALARY_END. For instance, user [admin,ford] *cannot* update the SALARY_AMOUNT column.

```
SQL> GRANT UPDATE ON COLUMN SALARY_HISTORY.SALARY_START
cont> TO [admin,ford];
SQL> GRANT UPDATE ON COLUMN SALARY_HISTORY.SALARY_END
cont> TO [admin,ford];
SQL> --
SQL> REVOKE UPDATE ON TABLE SALARY_HISTORY FROM [admin,ford];
SQL> --
SQL> COMMIT;
SQL> --
SQL> SHOW PROTECTION ON TABLE SALARY_HISTORY;
Protection on Table SALARY_HISTORY
  (IDENTIFIER=[grp2,jones],ACCESS=SELECT+INSERT+UPDATE+DELETE+SHOW+CREATE+
  ALTER+DROP+DBCTRL+DBADM+REFERENCES+SECURITY+DISTRIBTRAN)
  (IDENTIFIER=[*,*],ACCESS=NONE)
SQL> --
SQL> SHOW PROTECTION ON COLUMN SALARY_HISTORY.SALARY_START;
Protection on Column SALARY_HISTORY.SALARY_START
  (IDENTIFIER=[admin,ford],ACCESS=UPDATE)
  (IDENTIFIER=[*,*],ACCESS=NONE)
```

GRANT Statement

Example 4: Granting SELECT Privilege to All Users for a Sequence

```
SQL> SHOW PROTECTION ON SEQUENCE EMPID
Protection on Sequence EMPID
  (IDENTIFIER=[RDB,STRAUTS],ACCESS=SELECT+SHOW+ALTER+DROP+DBCTRL)
  (IDENTIFIER=[*,*],ACCESS=NONE)
SQL> GRANT SELECT ON SEQUENCE EMPID TO PUBLIC;
SQL> SHOW PROTECTION ON SEQUENCE EMPID;
Protection on Sequence EMPID
  (IDENTIFIER=[RDB,STRAUTS],ACCESS=SELECT+SHOW+ALTER+DROP+DBCTRL)
  (IDENTIFIER=[*,*],ACCESS=SELECT)
```

Example 5: Granting INSERT ON TABLE Privilege to a Role

```
SQL> SHOW PROTECTION ON TABLE JOBS
Protection on Table JOBS
  (IDENTIFIER=[250,254],ACCESS=SELECT+INSERT+UPDATE+DELETE+SHOW+CREATE+ALTER+
  DROP+DBCTRL+OPERATOR+DBADM+REFERENCES)
  (IDENTIFIER=PUBLIC,ACCESS=SELECT+INSERT+UPDATE+DELETE+SHOW+CREATE+ALTER+DROP
  +OPERATOR+DBADM+REFERENCES)
SQL> CREATE ROLE ADMINISTRATOR;
SQL> GRANT INSERT ON TABLE JOBS TO ADMINISTRATOR AFTER [250,254];
SQL> SHOW PROTECTION ON TABLE JOBS
Protection on Table JOBS
  (IDENTIFIER=[250,254],ACCESS=SELECT+INSERT+UPDATE+DELETE+SHOW+CREATE+ALTER+
  DROP+DBCTRL+OPERATOR+DBADM+REFERENCES)
  (IDENTIFIER=ADMINISTRATOR,ACCESS=INSERT)
  (IDENTIFIER=PUBLIC,ACCESS=SELECT+INSERT+UPDATE+DELETE+SHOW+CREATE+ALTER+DROP
  +OPERATOR+DBADM+REFERENCES)
```

Example 6: Allowing All Access to a User

```
SQL> -- Allow all access to user JAIN
SQL> GRANT SELECT ON DATABASE ALIAS *    to jain;
SQL> GRANT SELECT ON TABLE *           to jain;
SQL> GRANT EXECUTE ON MODULE *          to jain;
SQL> GRANT EXECUTE ON PROCEDURE *       to jain;
SQL> GRANT EXECUTE ON FUNCTION *        to jain;
```

GRANT Statement

Example 7: Automatically Creating a User While Granting Privileges

```
SQL> ATTACH 'FILENAME MF_PERSONNEL.RDB';
SQL> SHOW USERS
Users in database with filename mf_personnel.rdb
    tsmith
    jstuart
SQL> GRANT ALL ON DATABASE ALIAS RDB$DBHANDLE TO CDAY;
%RDB-W-META_WARN, metadata successfully updated with the reported warning
-RDMS-W-PRFCREATED, some users or roles were created
SQL> SHOW USERS
Users in database with filename mf_personnel.rdb
    tsmith
    jstuart
    cday
```

GRANT Statement: ANSI/ISO-Style

GRANT Statement: ANSI/ISO-Style

Creates or adds ANSI/ISO-style privileges to an entry of the Oracle Rdb access privilege set for a database, table, view, column, module, sequence, or routine. At database creation time, you specify whether the database protection mechanism will be ANSI/ISO-style or ACL-style. For more information on creating or changing the style of privileges associated with a database, see the CREATE DATABASE Statement.

Each entry in an ANSI/ISO-style access privilege set consists of an identifier and a list of privileges assigned to the identifier.

- Each identifier specifies a user or PUBLIC access.
- The set of privileges specifies what operations that user can perform on the database object.

ANSI/ISO-style privileges:

- Grant access to the creator when an object is created. Because only the creator is granted access to the newly created object, additional access must be granted explicitly.
- Support only the PUBLIC identifier as a wildcard.
- Support only user identifiers that translate to an OpenVMS user identification code (UIC), Rdb users, or roles.

For ANSI/ISO-style databases, a user's privileges are a combination of all privilege sets that apply to that user. The access privilege set is not order-dependent. The user matches the entry in the access privilege set; receives whatever privileges have been granted for the database object; and receives the privileges defined for PUBLIC. A user without an entry in the access privilege set receives only the privileges defined for PUBLIC, which always has an entry in the access privilege set even if PUBLIC has no access to the database, table, column, module, or external routine.

To remove privileges from or entirely delete an entry to the Oracle Rdb access privilege set for a database object, see the REVOKE Statement: ANSI/ISO-Style.

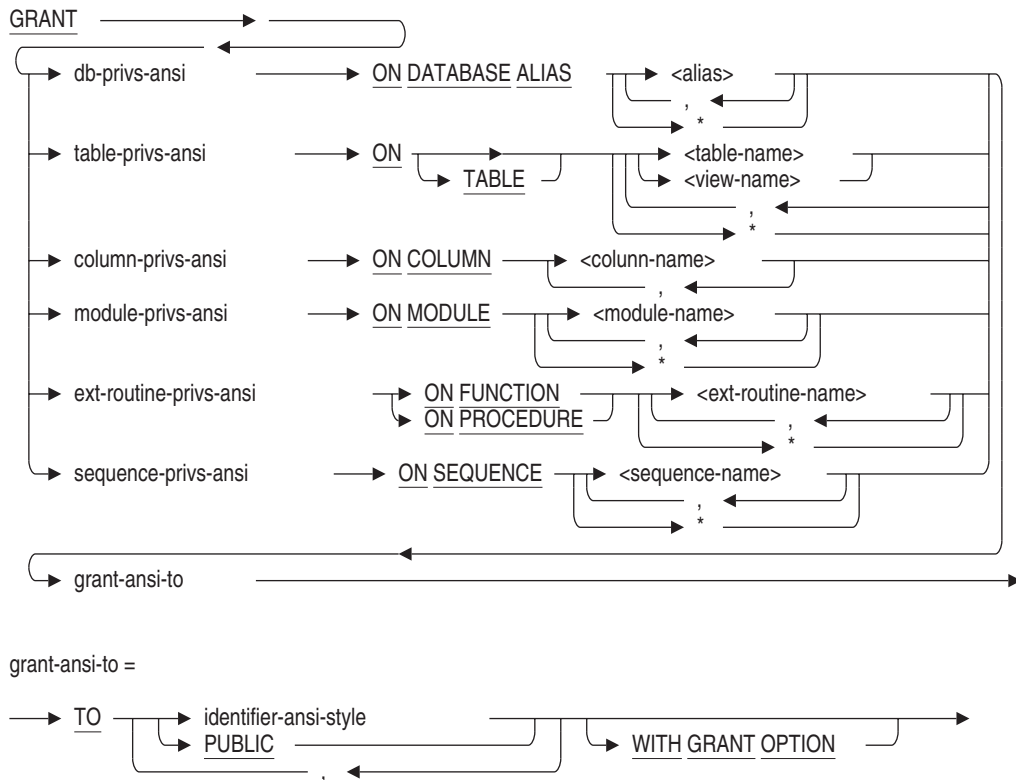
GRANT Statement: ANSI/ISO-Style

Environment

You can use the GRANT statement:

- In interactive SQL
- Embedded in host language programs to be precompiled
- As part of a nonstored procedure in a nonstored SQL module
- In dynamic SQL as a statement to be dynamically executed

Format



GRANT Statement: ANSI/ISO-Style

db-privs-ansi =

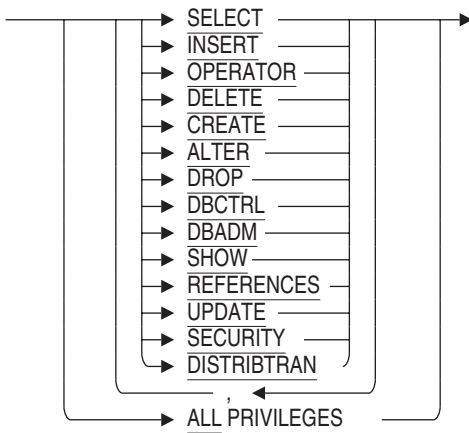
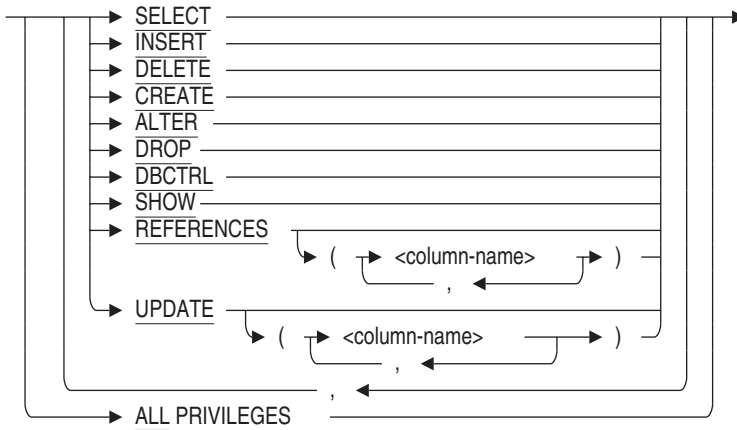


table-privs-ansi =



column-privs-ansi =



GRANT Statement: ANSI/ISO-Style

module-privs-ansi =



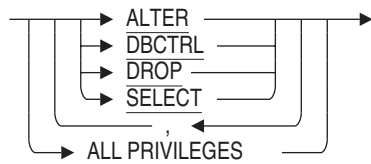
ext-routine-privs-ansi =



identifier-ansi-style =



sequence-privs-ansi =



Arguments

ALL PRIVILEGES

Specifies that SQL should grant all privileges to the specified users.

GRANT Statement: ANSI/ISO-Style

ON DATABASE ALIAS alias
ON TABLE table-name
ON COLUMN column-name
ON MODULE module-name
ON FUNCTION routine-name
ON PROCEDURE routine-name
ON SEQUENCE sequence-name

Specifies whether the GRANT statement applies to ACLs for the named object. You can specify a list of names for any form of the ON clause. You must qualify a column name with at least the associated table name.

ON DATABASE ALIAS *
ON TABLE *
ON MODULE *
ON FUNCTION *
ON PROCEDURE *
ON SEQUENCE *

Specifies whether the GRANT statement applies to ACLs for all objects of the specified type. If privileges are denied for the operation on some objects, then the GRANT is aborted.

db-privs-ansi
table-privs-ansi
column-privs-ansi
module-privs-ansi
ext-routine-privs-ansi

Specifies the set of privileges you want to add to an existing access privilege set entry or create in a new one. The operations permitted by a given privilege keyword differ, depending on whether you granted it for a database, table, column, module, or external routine. Table 7-5 lists the privilege keywords and their meanings for databases, tables, and columns.

The REFERENCES privilege lets you define a constraint for a database with ANSI/ISO-style privileges.

role-name

The name of a role, such as one created with the CREATE ROLE statement. If the role name exists as an operating system group or rights identifier, then Oracle Rdb will create the role automatically when you issue the GRANT statement. A role that is created automatically always has the attribute of IDENTIFIED EXTERNALLY.

GRANT Statement: ANSI/ISO-Style

TO identifier-ansi-style

Specifies the identifiers for the new or modified access privilege set entry. Specifying PUBLIC is equivalent to a wildcard specification of all user identifiers.

In ANSI/ISO-style databases, you are allowed to specify only single-user user identifiers; no general or system identifiers are allowed. Access privilege set entries identify only those users who are common to all groups defined by the individual identifiers. Users who do not match all identifiers are not controlled by that entry. ANSI/ISO-style access privilege sets support only user identifiers.

user-identifier

Specifies a user identifier that uniquely identifies each user on the system.

On OpenVMS, the user identifier consists of the standard OpenVMS user identification code (UIC), a group name and a member name (user name). The group name is optional. The user identifier can be in either numeric or alphanumeric format. The following are all valid user identifiers that could identify the same user:

```
K_JONES  
[SYSTEM3, K_JONES]  
[341,311]
```

When Oracle Rdb creates an ANSI/ISO-style database, the creator of the database gets all privileges, and the PUBLIC entry gets no privileges.

In an ANSI/ISO-style database, you cannot use multiple user identifiers.

In ANSI/ISO-style user identifiers, the only wildcard allowed is in the public identifier [`*`,`*`].

For more information about identifiers, see the OpenVMS operating system documentation.

WITH GRANT OPTION

Allows the user who has been granted a privilege the option of granting that privilege to other users.

The WITH GRANT OPTION clause specifies that the grantees in the TO clause may grant the privileges in the privilege list to other users for as long as they have the privileges. When the privilege is revoked from the grantee who received the privileges with the WITH GRANT OPTION clause, the privileges also are revoked from all the users who received the privileges from that grantee (unless these users have received the privilege from yet another user who still has the privilege).

GRANT Statement: ANSI/ISO-Style

Usage Notes

- For information on how to set up an ANSI/ISO-style database protection mechanism, see the CREATE DATABASE Statement.
- You can use the GRANT (ANSI/ISO-style) statement to modify existing access privilege set entries or create new ones.
To modify an existing access privilege set entry, specify the same identifier in the TO clause as in the existing entry.
To create a new access privilege set entry, specify an identifier that is not already part of an entry.
- Users with the DBADM database privilege can perform any data definition or data manipulation operation on any named object, including the database, regardless of the ACL for the object. The DBADM privilege is the most powerful privilege in Oracle Rdb because it can override most privilege checks performed by Oracle Rdb. Users with the DBADM database privilege implicitly receive *all* privileges for all objects, except the SECURITY database privilege.
- You must execute the GRANT (ANSI/ISO-style) statement in a read/write transaction. If you issue this statement when there is no active transaction, Oracle Rdb starts a read/write transaction implicitly.
- Privileges on a column are determined by the privileges defined on the table combined with those specified for the specific column access privilege set.
- A user with UPDATE or REFERENCES privilege on the table automatically receives the same privileges on all columns in the table. With UPDATE and REFERENCES privileges, you must have the privilege for either the column or the table to update a column. However, you can restrict UPDATE and REFERENCES privileges by defining them only on specific columns you want users to be able to update or define constraints for, and thus remove the privilege from the table entry.
You can modify the data in a column only if you have the UPDATE privilege for the column and the SELECT privilege for the database.
- The REFERENCES privilege lets you define a constraint.
- You cannot GRANT privileges on procedures or functions created as part of CREATE or ALTER MODULE.

GRANT Statement: ANSI/ISO-Style

- A user must have SELECT privileges on a sequence to use the NEXTVAL and CURRVAL pseudocolumns.
- If SECURITY CHECKING IS INTERNAL, Rdb will create an implicit CREATE USER and CREATE ROLE for any user-identifier or general-identifier not currently defined as a USER or ROLE.
- Unlike base tables, the data in temporary tables is not actually stored in the database, thus temporary tables never update the database.

Only the privileges associated with the temporary table are considered when performing security validation during data manipulation operations. For example, if a user can attach to the database (requires SELECT privilege only) and is granted INSERT privilege to a global or local temporary table, then the user (or an invoker's rights stored routine) can update the temporary table.

For more information on protection for an Oracle Rdb database, see the chapter on defining privileges in the *Oracle Rdb Guide to Database Design and Definition*.

Examples

Example 1: Using PUBLIC as a wildcard

This example shows that PUBLIC translates to [*,*] and can be used to grant access to the database for all users.

GRANT Statement: ANSI/ISO-Style

```
SQL> show protection on database rdb$dbhandle;
Protection on Alias RDB$DBHANDLE
[RDB,DOCS]:
  With Grant Option:      SELECT, INSERT, UPDATE, DELETE, SHOW, CREATE, ALTER, DROP,
                          DBCTRL, OPERATOR, DBADM, SECURITY, DISTRIBTRAN
  Without Grant Option:   NONE
[*,*]:
  With Grant Option:      NONE
  Without Grant Option:   NONE
SQL> grant select on database alias rdb$dbhandle to public;
SQL> show protection on database rdb$dbhandle;
Protection on Alias RDB$DBHANDLE
[RDB,DOCS]:
  With Grant Option:      SELECT, INSERT, UPDATE, DELETE, SHOW, CREATE, ALTER, DROP,
                          DBCTRL, OPERATOR, DBADM, SECURITY, DISTRIBTRAN
  Without Grant Option:   NONE
[*,*]:
  With Grant Option:      NONE
  Without Grant Option:   SELECT
SQL> commit;
```

Example 2: Granting a privilege with the WITH GRANT OPTION clause.

This example shows how the WITH GRANT OPTION causes Rdb to maintain a separate list of privileges that were granted by a user with ability to GRANT to others. This extra information is queried using the SHOW USERS WITH and SHOW USERS GRANTING commands.

GRANT Statement: ANSI/ISO-Style

```
SQL> show protection on table EMPLOYEES;
Protection on Table EMPLOYEES
[RDB,DOCS]:
  With Grant Option:      SELECT,INSERT,UPDATE,DELETE,SHOW,CREATE,ALTER,DROP,
                          DBCTRL,REFERENCES
  Without Grant Option:   NONE
[*,*]:
  With Grant Option:      NONE
  Without Grant Option:   NONE
SQL> grant delete on employees to freeman with grant option;
SQL>
SQL> show protection on table EMPLOYEES;
Protection on Table EMPLOYEES
[RDB,FREEMAN]:
  With Grant Option:      DELETE
  Without Grant Option:   NONE
[RDB,DOCS]:
  With Grant Option:      SELECT,INSERT,UPDATE,DELETE,SHOW,CREATE,ALTER,DROP,
                          DBCTRL,REFERENCES
  Without Grant Option:   NONE
[*,*]:
  With Grant Option:      NONE
  Without Grant Option:   NONE
SQL>
SQL> -- Show the list of users who will loose their DELETE
SQL> -- privilege if the privilege is taken away from DOCS
SQL>
SQL> show users with delete on employees from DOCS;
Users granted privileges on table EMPLOYEES by [RDB,DOCS]
[RDB,FREEMAN]
[RDB,DOCS]
[RDB,FREEMAN]
SQL>
SQL> -- Check if anyone on the list has given DELETE to anyone else
SQL>
SQL> show users granting delete on employees to PUBLIC;
Users granting privileges on table EMPLOYEES to [*,*]
No users found
SQL>
```

Example 3: Granting column privileges

This example shows the two forms of the GRANT column statement and the effects it has on the target columns.

GRANT Statement: ANSI/ISO-Style

```
SQL> -- First show existing column protections
SQL>
SQL> show protection on column CANDIDATES.FIRST_NAME;
Protection on Column CANDIDATES.FIRST_NAME
SQL> show protection on column CANDIDATES.CANDIDATE_STATUS;
Protection on Column CANDIDATES.CANDIDATE_STATUS
SQL>
SQL> -- Show alternate formats for the GRANT column statement
SQL>
SQL> grant update (CANDIDATE_STATUS) on table CANDIDATES to freeman;
SQL> grant update on column CANDIDATES.FIRST_NAME to freeman;
SQL>
SQL> -- Show the effects of the GRANT statements
SQL>
SQL> show protection on column candidates.FIRST_NAME;
Protection on Column CANDIDATES.FIRST_NAME
[RDB,FREEMAN]:
  With Grant Option:      NONE
  Without Grant Option:   UPDATE
[*,*]:
  With Grant Option:      NONE
  Without Grant Option:   NONE
SQL> show protection on column candidates.CANDIDATE_STATUS;
Protection on Column CANDIDATES.CANDIDATE_STATUS
[RDB,FREEMAN]:
  With Grant Option:      NONE
  Without Grant Option:   UPDATE
[*,*]:
  With Grant Option:      NONE
  Without Grant Option:   NONE
SQL>
```

Example 4: Granting privileges on a Sequence

This example shows that the set of privileges granted by ALL for sequences is a small subset of those used for other objects such as tables and views.

GRANT Statement: ANSI/ISO-Style

```
SQL> create sequence EMPLOYEE_ID_GEN;
SQL> grant all on sequence EMPLOYEE_ID_GEN to freeman;
SQL> grant select on sequence EMPLOYEE_ID_GEN to public;
SQL> show protection on sequence EMPLOYEE_ID_GEN;
Protection on Sequence EMPLOYEE_ID_GEN
[RDB,FREEMAN]:
  With Grant Option:      NONE
  Without Grant Option:  SELECT,SHOW,ALTER,DROP,DBCTRL,REFERENCES
[RDB,DOCS]:
  With Grant Option:      SELECT,SHOW,ALTER,DROP,DBCTRL,REFERENCES
  Without Grant Option:  NONE
[*,*]:
  With Grant Option:      NONE
  Without Grant Option:  SELECT
SQL>
```

GRANT Statement: Roles

GRANT Statement: Roles

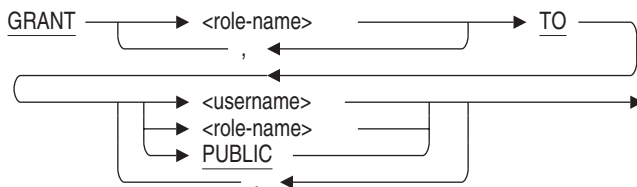
Grants a role to a user, another role, or the PUBLIC user.

Environment

You can use the GRANT statement for roles:

- In interactive SQL
- Embedded in host language programs to be precompiled
- As part of a nonstored procedure in a nonstored SQL module
- In dynamic SQL as a statement to be dynamically executed

Format



Arguments

role-name

The name of a role previously created with the CREATE ROLE statement. If the role name exists as an operating system group or rights identifier, then Oracle Rdb will automatically create the role when you issue the GRANT statement. A role that is created automatically always has the attribute IDENTIFIED EXTERNALLY.

TO username

TO role-name

TO PUBLIC

Specifies the user name, role name, or the PUBLIC user to which you want to grant the role. The PUBLIC user is the user name associated with all anonymous users who access the database.

GRANT Statement: Roles

If the user name or role name exists as an operating system user or rights identifier, then Oracle Rdb will automatically create the user name or role name when you issue the GRANT statement. A role or user that is created automatically is IDENTIFIED EXTERNALLY.

Usage Notes

- You must have the SECURITY privilege on the database to grant a role to a user or another role.
- The role being granted must not implicitly or explicitly be granted that role. This means that a role cannot be granted to itself.

Example

Example 1: Granting and Revoking Roles

```
SQL> -- Create three users and two roles. Oracle Rdb automatically
SQL> -- generates users and roles if they are identified externally.
SQL> CREATE USER ABLOWNEY IDENTIFIED EXTERNALLY;
SQL> CREATE USER BGREMO IDENTIFIED EXTERNALLY;
SQL> CREATE USER LWARD IDENTIFIED EXTERNALLY;
SQL> CREATE ROLE SALES_MANAGER IDENTIFIED EXTERNALLY;
SQL> CREATE ROLE DIVISION_MANAGER IDENTIFIED EXTERNALLY;
SQL> -- Grant the SALES_MANAGER role to users ABLOWNEY,
SQL> -- BGREMO, and to the DIVISION MANAGER ROLE.
SQL> GRANT SALES_MANAGER TO ABLOWNEY, BGREMO, DIVISION_MANAGER;
SQL> -- Grant the DIVISION_MANAGER role to LWARD. LWARD now
SQL> -- has both the SALES_MANAGER and DIVISION_MANAGER roles.
SQL> GRANT DIVISION_MANAGER TO LWARD;
SQL> -- Revoke the DIVISION_MANAGER role from LWARD. He has
SQL> -- left the company.
SQL> REVOKE DIVISION_MANAGER FROM LWARD;
SQL> -- Grant the DIVISION_MANAGER role to BGREMO. She
SQL> -- has been promoted to division manager.
SQL> GRANT DIVISION_MANAGER TO BGREMO;
```

Index

@ (at sign)

See EXECUTE statement

A

Access control lists (ACLs)

adding entries to, 7-371

changing, 7-371

creating entries, 7-371

database, 7-371

external routine, 7-371

general identifier, 7-376

module, 7-371

privileges, 7-377

system-defined identifier, 7-381

table, 7-371

user identifier, 7-371, 7-381

Access privilege sets

access control list (ACL) style, 7-371

adding entries to, 7-392

changing, 7-392

database

ANSI/ISO-style, 7-392

external routine, 7-392

module, 7-392

privileges, 7-377

table, 7-392

user identifier, 7-392, 7-397

ACLs

See Access control lists (ACLs)

Adding

entries to access privilege sets, 7-392

entries to ACLs, 7-371

AFTER clause

of GRANT statement, 7-374

Alias

for default database, 7-146, 7-382, 7-398

in CREATE TABLE statement, 7-63

in DECLARE ALIAS statement, 7-146

in GRANT statement, 7-376, 7-396

RDB\$DBHANDLE, 7-146, 7-382, 7-398

releasing, 7-249

ALIAS clause

of EXPORT statement, 7-332

ALIAS keyword

of DECLARE MODULE statement, 7-196

Allocating a snapshot page, 7-18

ALLOCATION clause

of CREATE STORAGE AREA clause, 7-13

ALTER privilege, 7-377

ANSI/ISO SQL standard

flagging violations of, 7-197

ANSI/ISO-style privileges, 7-392

ANSI_AUTHORIZATION qualifier

replaced by RIGHTS clause, 7-201

Assigning row caches, 7-13

At sign (@) command

See EXECUTE statement

Attaching to a database, 7-143

with DECLARE ALIAS statement, 7-143

Attach specifications

in DECLARE ALIAS statement, 7-147

in DROP DATABASE statement, 7-264

Authentication

user, 7-148, 7-264, 7-333

Authorization identifier
in precompiled SQL, 7–201
AUTHORIZATION keyword
of DECLARE MODULE statement, 7–197
of precompiled SQL, 7–197
AUTOMATIC clause, 7–63

B

BEGIN DECLARE statement
required terminators, 7–312
BYPASS privilege, 7–369, 7–383

C

CACHE USING clause
of CREATE STORAGE AREA clause, 7–13
Calculated columns
See COMPUTED BY columns
CALL statement
of CREATE TRIGGER statement, 7–110
CASCADE keyword
DROP SCHEMA statement, 7–286
DROP TABLE statement, 7–296
DROP VIEW statement, 7–305
of DROP INDEX statement, 7–269
of SQL module language, 7–269
Cascading delete
defining triggers, 7–121
schemas, 7–286
tables, 7–296
using triggers, 7–109
views, 7–305
Cascading update
using triggers, 7–109
Catalog
deleting, 7–255
DROP CATALOG statement, 7–255
CATALOG keyword
of DECLARE MODULE statement, 7–197
of SQL module language, 7–256
of SQL precompiler, 7–197
Character length
in precompiled SQL, 7–197, 7–198

CHARACTER LENGTH clause
in DECLARE MODULE statement, 7–197
in precompiled SQL, 7–197, 7–202
Character set
in precompiled SQL, 7–197, 7–200
in SQL precompiler, 7–199
national
of DECLARE ALIAS statement, 7–149
of DECLARE ALIAS statement, 7–147
of DECLARE MODULE statement, 7–199
of DEFAULT CHARACTER SET clause,
7–197

CHECK clause
of CREATE TABLE statement, 7–64
Check option clause, 7–134, 7–137, 7–139
CHECKSUM CALCULATION clause
of CREATE STORAGE AREA clause, 7–14

C language
dynamic SQL, 7–321
SQL module language, 7–321

Clumps
in CREATE STORAGE AREA clause, 7–17
COBOL language
dynamic SQL example, 7–327

COLLATING SEQUENCE clause
See also DROP COLLATING SEQUENCE
statement

Column
AUTOMATIC columns, 7–63
COMPUTED BY columns, 7–64
defining, 7–56
defining protection, 7–389
deleting, 7–296
DROP TABLE statement, 7–296
IDENTITY columns, 7–70

Column constraint
in CREATE TABLE statement, 7–64

Column default value, 7–67, 7–91

Column privileges, 7–377

Command line recall in interactive SQL, 7–310

Comments
adding to storage maps, 7–28

Compound statements
FOR (counted) control statement, 7–350
FOR control statement, 7–346

- Compressing
 - rows for partitioned tables, 7–28
- COMPUTED BY columns
 - defining in tables, 7–64, 7–65
 - deleting from tables, 7–64
 - with select expressions, 7–100
- Concurrency
 - See* Isolation level
- Connecting to a database
 - with DECLARE ALIAS statement, 7–143 to 7–154
- Consistency
 - See* Isolation level
- Constraint, 7–56
 - contrasting column and table constraints, 7–76
 - DEFERRABLE clause, 7–65
 - defining
 - in CREATE VIEW statement, 7–134
 - deleting, 7–261
 - naming in
 - CONSTRAINT clause, 7–66
 - CREATE TABLE statement, 7–64, 7–72
 - CREATE VIEW statement, 7–135
 - naming in DEFERRABLE clause, 7–64
 - NOT DEFERRABLE clause, 7–65
 - segmented strings and, 7–80
 - sequence of definition, 7–80
- CONSTRAINT clause
 - of CREATE TABLE statement, 7–66
 - to name constraints, 7–66
- Constraint definitions
 - adding to repository, 7–80
- Context file
 - DECLARE MODULE statement, 7–195
 - DECLARE TRANSACTION statement, 7–223
- Control statements
 - FOR, 7–346
 - FOR (counted), 7–350
- Correlation name
 - specifying in COMPUTED BY clause, 7–100
 - specifying in CREATE TRIGGER statement, 7–112
- COUNT function
 - specifying in COMPUTED BY clause, 7–100
- CREATE privilege, 7–377
- CREATE SEQUENCE statement, 7–2
 - CACHE clause, 7–3
 - CYCLE clause, 7–4
 - INCREMENT BY clause, 7–4
 - MAXVALUE clause, 7–4
 - MINVALUE clause, 7–5
 - ORDER clause, 7–5
 - RANDOMIZE clause, 7–5
 - START WITH clause, 7–6
 - STORED NAME clause, 7–6
 - WAIT clause, 7–7
- CREATE STORAGE AREA clause, 7–11
 - ALLOCATION clause, 7–13
 - CACHE USING clause, 7–13
 - CHECKSUM CALCULATION clause, 7–14
 - EXTENT clause, 7–15
 - FILENAME clause, 7–15
 - INTERVAL clause, 7–16
 - LOCKING clause, 7–16
 - NO ROW CACHE clause, 7–17
 - of CREATE DATABASE statement, 7–11
 - PAGE FORMAT clause, 7–17
 - PAGE SIZE clause, 7–18
 - RDB\$SYSTEM clause, 7–19
 - SNAPSHOT ALLOCATION clause, 7–18
 - SNAPSHOT EXTENT clause, 7–18
 - SNAPSHOT FILENAME clause, 7–18
 - THRESHOLDS clause, 7–19
- CREATE STORAGE MAP statement, 7–25
 - for table with data, 7–38
 - LOGGING clause, 7–29
 - NOLOGGING clause, 7–29
 - OTHERWISE IN clause, 7–30
 - PARTITIONING IS NOT UPDATABLE clause, 7–30
 - PARTITIONING IS UPDATABLE clause, 7–31
 - PARTITION name clause, 7–30
 - PLACEMENT VIA INDEX clause, 7–31
 - STORAGE MAP clause, 7–32
 - store clause, 7–32
 - STORED NAME IS clause, 7–34

CREATE STORAGE MAP statement (cont'd)

- STORE IN clause, 7-32
- STORE LISTS clause, 7-32
- STORE RANDOMLY ACROSS clause, 7-34
- STORE USING clause, 7-34
- THRESHOLDS clause, 7-34
- VERTICAL PARTITION name clause, 7-35
- WITH LIMIT OF clause, 7-35

CREATE SYNONYM statement, 7-52

- FOR clause, 7-53
- OR REPLACE clause, 7-53
- PUBLIC clause, 7-53
- synonym-name clause, 7-53

CREATE TABLE statement, 7-56

- AUTOMATIC clause, 7-63
- CHECK clause, 7-64
- column constraints, 7-64
- CONSTRAINT clause, 7-66
- DEFERRABLE clause, 7-65
- FROM path-name clause, 7-69
- IDENTITY clause, 7-70
- LIKE clause, 7-70
- maintaining referential integrity, 7-96
- naming constraints in, 7-66
- NOT DEFERRABLE clause, 7-65
- REFERENCES clause, 7-71
- STORED NAME IS clause, 7-72
- table constraints, 7-64, 7-72
- used in program
 - restriction, 7-80

CREATETAB privilege

- See also* CREATE privilege

CREATE TRIGGER statement, 7-109

- DELETE clause, 7-111
- description, 7-109
- environment, 7-110
- ERROR clause, 7-111
- FOR EACH ROW clause, 7-112
- in dynamic SQL, 7-110
- in embedded SQL, 7-110
- in interactive SQL, 7-110
- INSERT clause, 7-112
- REFERENCING clause, 7-112
- STORED NAME IS clause, 7-113
- triggered action clause, 7-113

CREATE TRIGGER statement (cont'd)

- UPDATE clause, 7-113
 - using a new correlation name, 7-112
 - using an old correlation name, 7-112

CREATE USER statement, 7-130

- ACCOUNT LOCK clause, 7-130
- COMMENT clause, 7-130
- IDENTIFIED EXTERNALLY clause, 7-131
- PROFILE clause, 7-131
- PUBLIC clause, 7-131

CREATE VIEW statement, 7-133, 7-142

- read-only views, 7-134, 7-136
- rules for updating views, 7-136
- STORED NAME IS clause, 7-135

Creating

- See also* Defining
- access privilege set entries, 7-392
- ACL entries, 7-371
- columns, 7-56
- cursors, 7-155, 7-172, 7-178
- storage areas, 7-11
- storage maps, 7-25, 7-32
- tables, 7-56
- triggers, 7-109
- views, 7-133

Cursor, 7-155, 7-172, 7-178

- See also* List cursor
- classes, 7-155
- declaring, 7-155, 7-172, 7-178
- declaring a holdable cursor, 7-171
- declaring a scrollable list cursor, 7-170
- dynamic
 - restriction, 7-174
- FETCH statement, 7-337
- holdable, 7-171
- insert-only, 7-156
- list, 7-156
- positioning, 7-339
- read-only, 7-156, 7-164
- restriction, 7-166
- result table created with OPEN, 7-155
- retrieving rows from, 7-337
- scope and extent, 7-155
- table, 7-156
- update-only, 7-156

Cursor (cont'd)
when result table exists, 7-155

D

Database

allocating pages, 7-13
allocating snapshot pages, 7-18
attaching to
with DECLARE ALIAS statement, 7-143
copying without data, 7-334
declaring, 7-143
default access, 7-371
default protection on, 7-371
deleting, 7-263
detaching, 7-249
duplicating without data, 7-334
granting access, 7-371
ANSI/ISO-style, 7-392
invoking, 7-143 to 7-154
page format, 7-17
page size, 7-18
privileges, 7-377
restricted access to, 7-151
restricting creation, 7-385
specifying
in GRANT statement, 7-376, 7-396
specifying extent pages, 7-18

Database access

restricted, 7-151

Database option

of DECLARE ALIAS statement, 7-146

Database privileges, 7-371, 7-377

ANSI/ISO-style, 7-392

DATA clause

of EXPORT statement, 7-334

Data manipulation statements

DELETE statement, 7-236

using with views, 7-136

Date format

DEFAULT DATE FORMAT clause, 7-198

specifying

in precompiled SQL, 7-198

Date-time data types

in CREATE TABLE statement, 7-67

DBADM privilege, 7-377, 7-383, 7-384, 7-398

DBCTRL privilege, 7-377

DBKEY SCOPE clause

of DECLARE ALIAS statement, 7-146

Deadlock

avoiding, 7-163

Deassigning row caches, 7-17

DECIMAL data type

conversion by CREATE TABLE, 7-93

DECLARE ALIAS statement, 7-143

attach specifications, 7-147

database option, 7-146

DBKEY SCOPE clause, 7-146

-dbtype option, 7-146

default alias, 7-146

default character set, 7-147

DEFAULT CHARACTER SET clause, 7-147

environment, 7-143

FILENAME clause, 7-147

in embedded SQL, 7-143

in SQL module language procedures, 7-143

MULTISHEMA IS clause, 7-149

national character set, 7-149

NATIONAL CHARACTER SET clause, 7-149

PATHNAME clause, 7-150

PRESTARTED TRANSACTIONS clause,
7-150

repository path names, 7-150

RESTRICTED ACCESS clause, 7-151

ROWID SCOPE clause, 7-151

DECLARE CURSOR statement, 7-155

FOR UPDATE clause, 7-160

read-only cursors, 7-164

SCROLL keyword, 7-157, 7-162

WHERE CURRENT OF clause, 7-163

WITH HOLD clause, 7-163

DECLARE DEFAULT TRANSACTION, 7-225

DECLARE FUNCTION statement, 7-185

DECLARE LOCAL TEMPORARY TABLE

statement, 7-186

creating, 7-186

deleting rows on commit, 7-190

preserving rows on commit, 7-190

DECLARE LOCAL TEMPORARY TABLE
 statement (cont'd)
 restrictions, 7-190
 virtual memory requirements, 7-190

DECLARE MODULE statement, 7-195
 ALIAS keyword, 7-196
 AUTHORIZATION keyword, 7-197
 CATALOG keyword, 7-197
 character set, 7-199
 context files, 7-195
 DEFAULT CHARACTER SET clause, 7-197
 DEFAULT DATE FORMAT clause, 7-198
 DIALECT clause, 7-198
 environment, 7-195
 in embedded SQL, 7-195
 in precompiled SQL, 7-202
 KEYWORD RULES clause, 7-199
 LITERAL CHARACTER SET clause, 7-199
 MODULE keyword, 7-200
 NAMES ARE clause, 7-200
 national character set, 7-200
 PARAMETER COLONS clause, 7-200
 QUOTING RULES clause, 7-200
 RIGHTS clause, 7-201
 SCHEMA keyword, 7-201
 VIEW UPDATE RULES clause, 7-201

DECLARE PROCEDURE statement, 7-205

DECLARE Routine statement, 7-206
 environment, 7-206
 format, 7-206
 in context files, 7-206
 in embedded SQL, 7-206
 in interactive SQL, 7-206

DECLARE STATEMENT statement, 7-213

DECLARE TABLE statement, 7-215

DECLARE TRANSACTION statement, 7-222
 contrasted with SET TRANSACTION
 statement, 7-222, 7-226
 defaults, 7-225
 dynamically executed, 7-228
 environment, 7-223
 format, 7-223
 in embedded SQL, 7-223
 in interactive SQL, 7-223
 isolation level option in, 7-223

DECLARE TRANSACTION statement (cont'd)
 multiple, in programs, 7-228
 restriction in programs, 7-228

DECLARE variable statement, 7-233

Declaring
 holdable cursor, 7-171
 scrollable list cursor, 7-170

Declaring a database, 7-143
 with DECLARE ALIAS statement, 7-143 to
 7-154

Default character set
 in precompiled SQL, 7-197
 of DECLARE ALIAS statement, 7-147

DEFAULT CHARACTER SET clause
 in precompiled SQL, 7-197, 7-202
 of DECLARE ALIAS statement, 7-147
 of DECLARE MODULE statement, 7-197

Default database
 with DECLARE ALIAS statement, 7-146

DEFAULT DATE FORMAT clause
 in precompiled SQL, 7-198
 of DECLARE MODULE statement, 7-198

Default value
 specifying
 in ALTER TABLE statement, 7-67
 in CREATE TABLE statement, 7-91

DEFERRABLE clause
 constraints, 7-64
 of CREATE TABLE statement, 7-65

Defining
See Creating
 access privilege set entries, 7-392
 database access, 7-371
 ANSI/ISO-style, 7-392
 external routine access, 7-371, 7-392
 module access, 7-371, 7-392
 privileges, 7-371, 7-392
 table access, 7-371, 7-392

DELETE privilege, 7-377

DELETE statement, 7-236
 in CREATE TRIGGER statement, 7-111
 OPTIMIZE WITH clause, 7-238
 specifying through
 CREATE TRIGGER statement, 7-109
 using with views, 7-136

- Deleting
 - catalogs, 7-255
 - constraints, 7-261
 - databases, 7-263
 - data in views, 7-136
 - data with DELETE statement, 7-236
 - external functions, 7-282, 7-283
 - external procedures, 7-282, 7-283
 - external routines, 7-283
 - indexes, 7-269
 - modules, 7-271
 - repository definitions, 7-276
 - rows of tables, 7-236
 - schemas, 7-286
 - storage maps, 7-292
 - stored functions, 7-282, 7-283
 - stored procedures, 7-282, 7-283
 - stored routines, 7-283
 - tables, 7-296
 - triggers, 7-301
 - views, 7-305
- Deleting a query outline, 7-274
- Deprecated feature
 - SQLOPTIONS=ANSI_AUTHORIZATION, 7-201
 - SQLOPTIONS=ANSI_DATE, 7-198
 - SQLOPTIONS=ANSI_IDENTIFIERS, 7-199
 - SQLOPTIONS=ANSI_PARAMETERS, 7-200
 - SQLOPTIONS=ANSI_QUOTING, 7-200
- DESCRIBE statement, 7-242, 7-245
 - in a PL/I program, 7-245
 - INPUT clause, 7-243
 - INTO clause, 7-243
 - manual alternative to, 7-244
 - MARKERS clause, 7-243
 - OUTPUT clause, 7-244
 - overriding values set by, 7-244
 - SELECT LIST clause, 7-244
 - SQLDA, 7-243
 - statement-name, 7-244
- DESCRIPTOR clause
 - EXECUTE statement, 7-319
- Detaching from databases, 7-249
- DIALECT clause
 - See also* SET DIALECT statement in Volume 3
 - in precompiled SQL, 7-198
 - of DECLARE MODULE statement, 7-198
- Dialect setting
 - in precompiled SQL, 7-198
 - of DECLARE MODULE statement, 7-198
- DISCONNECT statement, 7-249
- DISTRIBTRAN privilege, 7-377
- DROP CATALOG statement, 7-255
 - environment, 7-255
 - in dynamic SQL, 7-255
 - in embedded SQL, 7-255
 - in interactive SQL, 7-255
- DROP COLLATING SEQUENCE statement, 7-258
- DROP CONSTRAINT statement, 7-261
- DROP DATABASE statement, 7-263
 - attach specification, 7-264
 - environment, 7-263
 - in dynamic SQL, 7-263
 - in embedded SQL, 7-263
 - in interactive SQL, 7-263
- DROP DOMAIN statement, 7-266
 - See also* ALTER DOMAIN statement in Volume 2
 - See also* CREATE DOMAIN statement in Volume 2
- DROP FUNCTION statement, 7-282
- DROP INDEX statement, 7-269
- DROP MODULE statement, 7-271
- DROP OUTLINE statement, 7-274
- DROP PATHNAME statement, 7-276
- DROP privilege, 7-377
- DROP PROCEDURE statement, 7-282
- DROP ROLE statement, 7-280
 - CASCADE clause, 7-280
 - IF EXISTS clause, 7-280
 - RESTRICT clause, 7-280
 - role-name clause, 7-280
- Drop Routine statement
 - See* DROP FUNCTION statement, DROP PROCEDURE statement

- DROP SCHEMA statement, 7-286
 - CASCADE keyword, 7-286
 - environment, 7-286
 - in dynamic SQL, 7-286
 - in embedded SQL, 7-286
 - in interactive SQL, 7-286
 - RESTRICT keyword, 7-286
- DROP SEQUENCE statement, 7-289
 - CASCADE clause, 7-289
 - IF EXISTS clause, 7-289
 - RESTRICT clause, 7-289
 - sequence-name clause, 7-289
- DROP statement
 - general usage notes, 7-254
- DROP STORAGE MAP statement, 7-292
- DROP SYNONYM statement, 7-294
 - CASCADE clause, 7-294
 - IF EXISTS clause, 7-294
 - PUBLIC clause, 7-294
 - RESTRICT clause, 7-295
 - synonym-name clause, 7-295
- DROP TABLE statement, 7-296
 - CASCADE keyword, 7-296
 - RESTRICT keyword, 7-296
- DROP TRIGGER statement, 7-301
 - arguments, 7-301
 - environment, 7-301
 - in embedded SQL, 7-301
 - in interactive SQL, 7-301
 - restrictions, 7-301
- DROP USER statement, 7-303
 - CASCADE clause, 7-303
 - RESTRICT clause, 7-303
 - username clause, 7-303
- DROP VIEW statement, 7-305
 - CASCADE keyword, 7-305
 - RESTRICT keyword, 7-305
- Dynamic DECLARE CURSOR statement, 7-172
 - accessing views
 - restriction, 7-174
- Dynamic SQL, 7-242
 - C, 7-321
 - COBOL, 7-327
 - CREATE TRIGGER statement, 7-110
 - DECLARE STATEMENT statement, 7-213

- Dynamic SQL (cont'd)
 - DESCRIBE statement, 7-242
 - EXECUTE IMMEDIATE statement, 7-326
 - EXECUTE statement, 7-318
 - multiple
 - DECLARE TRANSACTION statements, 7-228
 - parameter markers, 7-242
 - PL/I, 7-245
 - select lists, 7-242
 - SQL module language, 7-321
 - statement names, 7-213, 7-244, 7-318

E

- EDIT statement, 7-308
 - invoking DECTPU, 7-308
 - invoking EDT, 7-308
 - invoking LSE, 7-308
 - SQL\$EDIT logical name, 7-308
- EDT editor
 - editing in interactive SQL, 7-308
- Embedding SQL statements in programs
 - See* SQL precompiler
- END DECLARE statement, 7-312, 7-313
 - required terminators, 7-312
- Erasing data
 - DELETE statement, 7-236
 - in views, 7-136
- ERROR clause
 - of CREATE TRIGGER statement, 7-111
- Error message
 - generated by CREATE TRIGGER statement, 7-111
 - MAXRELVER, 7-80
 - MAXVIEWID, 7-138
- EVE
 - editing in interactive SQL, 7-308
- EXCEPTION clause
 - GET DIAGNOSTICS statement, 7-359
- Execute (@) statement for running command files, 7-315

- EXECUTE IMMEDIATE statement, 7–326, 7–327
 - in a COBOL program, 7–327
 - parameter, 7–326
 - statement string, 7–326
- EXECUTE privilege, 7–377
- EXECUTE statement, 7–318
 - @ (at sign) command, 7–315
 - parameter, 7–319
 - parameter markers, 7–319, 7–320
 - SQLCA, 7–320
 - statement-name, 7–319
 - USING DESCRIPTOR clause, 7–320
- EXIT statement, 7–331
- Exporting a database
 - restriction, 7–335
- EXPORT statement, 7–332
 - ALIAS clause, 7–332
 - contrasted with IMPORT statement, 7–332
 - DATA clause, 7–334
 - EXTENSIONS clause, 7–334
 - extents
 - restriction, 7–336
 - FILENAME clause, 7–332
 - INTO clause, 7–333
 - PATHNAME clause, 7–332
 - .rbr file, 7–333
 - restriction, 7–335
- Extended dynamic DECLARE CURSOR statement, 7–178
- EXTENSIONS clause
 - of EXPORT statement, 7–334
- EXTENT clause
 - of CREATE STORAGE AREA clause, 7–15
- Extent page, specifying, 7–18
- External functions
 - creating, 7–185
 - deleting, 7–282, 7–283
 - restrict deleting, 7–283
- External procedure
 - declaring, 7–205
 - deleting, 7–282, 7–283
 - restrict deleting, 7–283

- External routine
 - default protection on, 7–371
 - deleting, 7–282
 - EXECUTE privilege, 7–377
 - granting access, 7–371, 7–392
 - granting privilege, 7–371
 - privileges, 7–371, 7–392
 - restrict deleting, 7–283
 - specifying
 - in GRANT statement, 7–376, 7–396

F

- FETCH statement, 7–337
- FILENAME clause
 - DECLARE ALIAS statement, 7–147
 - of CREATE STORAGE AREA clause, 7–15
 - of EXPORT statement, 7–332
- Filling storage areas, 7–29
- FINISH statement
 - See* DISCONNECT statement
- FOR (counted) control statement
 - DO clause, 7–351
 - of compound statement, 7–350
- FOR control statement
 - AS EACH ROW OF clause, 7–346, 7–347
 - beginning label, 7–347, 7–351
 - DO clause, 7–347
 - ending label, 7–347, 7–351
 - FOR variable name clause, 7–347, 7–351
 - of compound statement, 7–346
- FOR EACH ROW clause
 - of CREATE TRIGGER statement, 7–112
- FOR EACH STATEMENT clause
 - of CREATE TRIGGER statement, 7–112
- FOR UPDATE clause, 7–160
- FROM path-name clause
 - in CREATE TABLE statement, 7–69
- Function
 - See also* External routine, Stored Function

G

General identifiers, 7-376
GET DIAGNOSTICS statement, 7-357
 EXCEPTION clause, 7-359
 LIMIT_CPU_TIME clause, 7-360
 LIMIT_ELAPSED_TIME clause, 7-360
 LIMIT_ROWS_FETCHED clause, 7-360
 RETURNED_SQLCODE clause, 7-359
 RETURNED_SQLSTATE clause, 7-359
 statement-item-name, 7-359
GET ENVIRONMENT statement, 7-366
Getting out of interactive SQL
 EXIT statement, 7-331
Global temporary table
 See Temporary table
GRANT statement, 7-371
 AFTER clause, 7-374
 ANSI/ISO style, 7-392
 creating new access privilege set entries,
 7-398
 creating new ACL entries, 7-382
 database access, 7-371
 ANSI/ISO-style, 7-392
 external routine access, 7-371, 7-392
 general usage notes, 7-369
 modifying existing access privilege set entries,
 7-398
 modifying existing ACL entries, 7-382
 module access, 7-371, 7-392
 ON COLUMN clause, 7-376, 7-396
 ON DATABASE clause, 7-376, 7-396
 ON FUNCTION clause, 7-376, 7-396
 ON MODULE clause, 7-376, 7-396
 ON PROCEDURE clause, 7-376, 7-396
 ON SEQUENCE clause, 7-376, 7-396
 ON TABLE clause, 7-376, 7-396
 POSITION clause, 7-374
 privileges, 7-377
 RDB\$DBHANDLE default alias, 7-382,
 7-398
 roles, 7-404
 table access, 7-371, 7-392
 TO clause, 7-380

GRANT statement
 TO clause (cont'd)
 ANSI/ISO-style, 7-397
GROUP BY clause
 accessing view containing
 restriction, 7-174

H

Hidden delete
 See Cascading delete
Holdable cursor, 7-163
 declaring, 7-171
Horizontal partitioning, 7-35

I

Identifier character set
 in SQL precompiler, 7-199
IDENTIFIER CHARACTER SET clause
 in SQL precompiler, 7-199
Identifiers in access privilege sets, 7-392, 7-397
 in ANSI/ISO-style GRANT statement, 7-397
 multiple, 7-397
Identifiers in ACLs, 7-371
 general, 7-376
 in GRANT statement, 7-380
 multiple, 7-380
 system, 7-381
 user identifier, 7-381
IF EXISTS keyword
 of DROP INDEX statement, 7-269
 of SQL module language, 7-256, 7-258,
 7-261, 7-266, 7-269, 7-271, 7-274, 7-277,
 7-280, 7-282, 7-286, 7-289, 7-292, 7-294,
 7-296, 7-301, 7-305
Index
 deleting, 7-269
 DROP INDEX statement, 7-269
Initializing interactive SQL, 7-315
INPUT clause
 DESCRIBE statement, 7-243
Input parameter, 7-243
INSERT clause
 of CREATE TRIGGER statement, 7-112

INSERT privilege, 7-377
INSERT statement
 in a COBOL program, 7-327
 in CREATE TRIGGER statement, 7-112
 in dynamic SQL, 7-327
 parameter markers, 7-327
 specifying through
 CREATE TRIGGER statement, 7-109
 using with views, 7-136
Interactive SQL interface
 command line recall, 7-310
Internationalization features
 See also DROP COLLATING SEQUENCE
 statement
INTERVAL clause
 of CREATE STORAGE AREA clause, 7-16
INTO clause
 EXECUTE statement, 7-319
 of DESCRIBE statement, 7-243
 of EXPORT statement, 7-333
Isolation level
 in DECLARE TRANSACTION statement,
 7-223

K

Keyword
 controlling interpretation of
 in precompiled SQL, 7-198, 7-199
KEYWORD RULES clause
 in precompiled SQL, 7-199
 of DECLARE MODULE statement, 7-199

L

Language-Sensitive Editor
 See LSE
Leaving interactive SQL
 EXIT statement, 7-331
Length
 character
 in precompiled SQL, 7-197, 7-198
Limits and parameters
 maximum for segmented string fields, 7-335
 maximum number of tables, 7-80

Limits and parameters (cont'd)
 maximum number of views, 7-138
LIMIT_CPU_TIME clause
 GET DIAGNOSTICS statement, 7-360
LIMIT_ELAPSED_TIME clause
 GET DIAGNOSTICS statement, 7-360
LIMIT_ROWS_FETCHED clause
 GET DIAGNOSTICS statement, 7-360
List
 accessing with cursors, 7-156
 creating, 7-99
 filling storage areas
 randomly, 7-29
 sequentially, 7-29
 processing with host programs, 7-155
 setting a default storage area, 7-32
 storing in multiple storage areas, 7-32
 storing separately from table rows, 7-36
List cursor
 See also Cursor
 restriction, 7-166
 retrieving list segments from, 7-337
 scrollable, 7-157, 7-170
LITERAL CHARACTER SET clause
 of DECLARE MODULE statement, 7-199
Local temporary table
 See also Temporary table
 declaring explicitly, 7-186
Lock conflicts
 reducing, 7-163
LOCKING clause
 of CREATE STORAGE AREA clause, 7-16
LOGGING clause
 CREATE STORAGE MAP statement, 7-29
Logical area threshold, 7-34
Logical name
 RDBVMS\$CREATE_DB, 7-385
 SQL\$EDIT, 7-308
 SQLINI, 7-315
LSE
 editing in interactive SQL, 7-308
 invoking within SQL, 7-309
LSE\$ENVIRONMENT
 defining, 7-309

M

MARKERS clause of DESCRIBE statement, 7-243

Modifying

- access privilege set entries, 7-392
- ACL entries, 7-371
- data in views, 7-136

Module

- default protection on, 7-371
- deleting, 7-271
- EXECUTE privilege, 7-377
- granting access, 7-371, 7-392
- privileges, 7-371, 7-392
- specifying
 - in GRANT statement, 7-376, 7-396

MODULE keyword

- of DECLARE MODULE statement, 7-200

Multiple identifiers, 7-380

- not allowed in ANSI/ISO-style databases, 7-397

Multiple transaction declarations in programs, 7-228

MULTISCHHEMA IS ON clause

- in DECLARE ALIAS statement, 7-149

Multistatement procedure

- See also* Compound statement

N

Name

- character set for
 - precompiled SQL, 7-200
- dynamic SQL statements, 7-244
- statement (dynamic), 7-244

NAMES ARE clause

- in precompiled SQL, 7-202
- of DECLARE MODULE statement, 7-200

Naming a query, 7-160, 7-237

Naming constraints

- in CONSTRAINT clause, 7-66
- in CREATE TABLE statement, 7-64

National character set

- in precompiled SQL, 7-200
- of DECLARE ALIAS statement, 7-149
- of DECLARE MODULE statement, 7-200

NATIONAL CHARACTER SET clause

- in precompiled SQL, 7-200, 7-202
- of DECLARE ALIAS statement, 7-149

New-correlation-name

- using in CREATE TRIGGER statement, 7-112

NOLOGGING clause

- CREATE STORAGE MAP statement, 7-29

NO ROW CACHE clause

- of CREATE STORAGE AREA clause, 7-17

NOT DEFERRABLE clause

- constraints, 7-64
- of CREATE TABLE statement, 7-65

O

Old-correlation-name

- using in CREATE TRIGGER, 7-112

OPER privilege, 7-369

OPTIMIZE clause

- AS keyword, 7-160, 7-237
- USING keyword, 7-161, 7-237

Optimizing

- queries, 7-161, 7-237
- using an outline, 7-161, 7-237
- using an query name, 7-160, 7-237

Optional predicate

- specifying through
 - CREATE TRIGGER statement, 7-109

Oracle Rdb databases

- specifying in DECLARE ALIAS statement, 7-146

OTHERWISE IN clause of CREATE STORAGE MAP statement, 7-30

Outline name

- using, 7-161, 7-237

OUTPUT clause

- of DESCRIBE statement, 7-244

Output parameter, 7-244

Overflow partition
in storage map definition, 7-30

P

PAGE FORMAT clause
of CREATE STORAGE AREA clause, 7-17

PAGE SIZE clause
of CREATE STORAGE AREA clause, 7-18

Parameter
in EXECUTE statement, 7-319
specifying dynamic statements, 7-326

PARAMETER COLONS clause
of DECLARE MODULE statement, 7-200

Parameter markers, 7-319, 7-320
DESCRIBE statement, 7-243
information in SQLDA, 7-242

Partitioning, 7-30, 7-35
strict, 7-30

PARTITIONING IS NOT UPDATABLE clause
CREATE STORAGE MAP statement, 7-30

PARTITIONING IS UPDATABLE clause
CREATE STORAGE MAP statement, 7-31

Partitioning tables
horizontally, 7-35
vertically, 7-32

PARTITION name clause
CREATE STORAGE MAP statement, 7-30

PATHNAME clause
DECLARE ALIAS statement, 7-150
of EXPORT statement, 7-332

Performance
optimizing queries, 7-161, 7-237

Persistent cursor, 7-163

PL/I language
dynamic SQL, 7-245

PLACEMENT VIA INDEX clause
of CREATE STORAGE MAP statement, 7-31

POSITION clause
of GRANT statement, 7-374

Positioning cursors, 7-339

Precompiled SQL
See SQL precompiler

Predicate
in CREATE TRIGGER statement, 7-113

Prestarted transaction
disabling, 7-150

PRESTARTED TRANSACTIONS clause
of DECLARE ALIAS statement, 7-150

Privilege
See also Protection
adding, 7-371, 7-392
ALTER, 7-377
ANSI/ISO-style
defining, 7-392
BYPASS, 7-369
CREATE, 7-377
database, 7-371
ANSI/ISO-style, 7-392
DBADM, 7-377, 7-384, 7-398
DBCTRL, 7-377
defining, 7-371
DELETE, 7-377
DISTRIBTRAN, 7-377
DROP, 7-377
EXECUTE, 7-377
external routine, 7-371, 7-392
GRANT statement, 7-371
ANSI/ISO-style, 7-392
INSERT, 7-377
module, 7-371, 7-392
OPER, 7-369
overridden
by BYPASS, 7-383
by DBADM, 7-383
by READALL, 7-383
by SECURITY, 7-383
by SYSPRV, 7-383
REFERENCES, 7-377
role-oriented, 7-383
SECURITY, 7-369, 7-377, 7-384
SELECT, 7-377
SHOW, 7-377
SYSPRV, 7-369
table, 7-371, 7-392
UPDATE, 7-377

Protection

See also Privilege
column level, 7-389

Q

Query naming, 7-160, 7-237

Query optimizer, 7-161, 7-237

Query outlines

deleting, 7-274

Quotation mark

controlling interpretation of

in precompiled SQL, 7-198, 7-200

QUOTING RULES clause

in precompiled SQL, 7-200

of DECLARE MODULE statement, 7-200

R

RDB\$DBHANDLE default alias, 7-146

in DECLARE ALIAS statement, 7-146

in GRANT statement, 7-382, 7-398

RDB\$SYSTEM clause

of CREATE STORAGE AREA clause, 7-19

RDB\$SYSTEM storage area, 7-19, 7-32

RDBVMS\$CREATE_DB logical name, 7-385

RDBVMS\$CREATE_DB rights identifier, 7-385

READALL privilege, 7-369, 7-383

Read-only cursor, 7-164

Read-only view, 7-136

restrictions, 7-134, 7-136

REFERENCES clause

of CREATE TABLE statement, 7-71

REFERENCES privilege, 7-377

REFERENCING clause

of CREATE TRIGGER statement, 7-112

Referencing table, 7-69, 7-71

Referential integrity

maintaining with

CREATE TABLE statement, 7-79, 7-96

CREATE TRIGGER statement, 7-109

table-specific constraints, 7-79

Relation

See Table

Removing data

DELETE statement, 7-236

in views, 7-136

Repository

adding constraint definitions to, 7-80

adding table definitions to, 7-80

adding trigger definitions to, 7-80

definitions

deleting, 7-276

path names

creating a table from a path name, 7-69

in CREATE TABLE statement, 7-69,

7-80

in DECLARE ALIAS statement, 7-150

using when deleting repository definitions,

7-276

RESTRICTED ACCESS clause

of DECLARE ALIAS statement, 7-151

Restricted access to database, 7-151

Restricting delete

schemas, 7-286

Restriction

CREATE STORAGE AREA clause, 7-17

CREATE TABLE statement

repository record structures, 7-73

used in program, 7-80

cursors

dynamic, 7-174

DECLARE CURSOR statement, 7-166

declared local temporary table, 7-190

DECLARE variable statement, 7-234

dynamic DECLARE CURSOR statement

accessing views, 7-174

exporting a database, 7-335

EXPORT statement, 7-335

extents, 7-336

GROUP BY clause

accessing view containing, 7-174

on cursors, 7-166

page-level locking, 7-17

row-level locking, 7-17

UNION clause

accessing view containing, 7-174

views

accessing using dynamic cursors, 7-174

Restriction (cont'd)
 WITH NO EXTENSIONS clause, 7-335
 RESTRICT keyword
 DROP SCHEMA statement, 7-286
 DROP TABLE statement, 7-296
 DROP VIEW statement, 7-305
 of DROP INDEX statement, 7-269
 of SQL module language, 7-269
 Result tables
 for cursors, 7-155, 7-172, 7-178
 Retrieving
 list segments, 7-337
 rows of cursors, 7-337
 RETURNED_SQLCODE clause
 GET DIAGNOSTICS statement, 7-359
 RETURNED_SQLSTATE clause
 GET DIAGNOSTICS statement, 7-359
 RIGHTS clause
 in precompiled SQL, 7-201
 of DECLARE MODULE statement, 7-201
 Role-oriented privilege, 7-383
 Roles
 GRANT statement, 7-404
 Row cache
 assignment, 7-13
 deassignment, 7-17
 ROWID SCOPE clause
 of DECLARE ALIAS statement, 7-151
 Running command files with execute (@)
 statement, 7-315
 RUNTIME option
 for DECLARE ALIAS statement, 7-151
 run-time string
 for DECLARE ALIAS statement, 7-151

S

Schema
 cascading delete, 7-286
 deleting, 7-286
 DROP SCHEMA statement, 7-286
 restricting delete, 7-286
 SCHEMA keyword
 of DECLARE MODULE statement, 7-201
 of SQL module language, 7-287
 of SQL precompiler, 7-201

Scratch table
See Declared local temporary table
 Scrollable list cursor, 7-157
 declaring, 7-170
 SCROLL keyword
 of DECLARE CURSOR statement, 7-157,
 7-162
 Security functions of triggers, 7-109
 SECURITY privilege, 7-369, 7-377, 7-383,
 7-384
 Segmented string
See also List
 Select expressions
 in COMPUTED BY clauses, 7-100
 SELECT LIST clause
 of DESCRIBE statement, 7-244
 Select lists
 DESCRIBE statement, 7-242, 7-244
 information in SQLDA, 7-244
 SELECT privilege, 7-377
 SELECT statement
 example in dynamic SQL, 7-245
 in a PL/I program, 7-245
 SET TRANSACTION statement
 contrasted with DECLARE TRANSACTION
 statement, 7-227
 SHOW privilege, 7-377
 SIGNAL statement
 of CREATE TRIGGER statement, 7-110
 SNAPSHOT ALLOCATION clause
 of CREATE STORAGE AREA clause, 7-18
 SNAPSHOT EXTENT clause
 of CREATE STORAGE AREA clause, 7-18
 SNAPSHOT FILENAME clause
 of CREATE STORAGE AREA clause, 7-18
 Snapshot page
 allocating, 7-18
 SQL\$EDIT logical name, 7-308
 SQLCA
 in EXECUTE statement, 7-320
 SQL command lines, editing with EDIT
 statement, 7-308
 SQLDA, 7-242, 7-339
 in DESCRIBE statement, 7-242, 7-243
 parameter markers, 7-242

SQLDA (cont'd)

select lists, 7-242

SQLINI command file, 7-315

SQL mapping routine

CREATE STORAGE MAP statement, 7-36

SQL module language

CASCADE keyword, 7-269

CATALOG keyword, 7-256

dynamic SQL example in C, 7-321

IF EXISTS keyword, 7-256, 7-258, 7-261,

7-266, 7-269, 7-271, 7-274, 7-277, 7-280,

7-282, 7-286, 7-289, 7-292, 7-294, 7-296,

7-301, 7-305

RESTRICT keyword, 7-269

SCHEMA keyword, 7-287

SQL precompiler

ALIAS keyword, 7-196

authorization identifier, 7-201

AUTHORIZATION keyword, 7-197

CATALOG keyword, 7-197

CHARACTER LENGTH clause, 7-197, 7-202

character set, 7-197, 7-199, 7-200

DECLARE MODULE statement, 7-195,
7-202

default character set, 7-197

DEFAULT CHARACTER SET clause, 7-197,
7-202

DEFAULT DATE FORMAT clause, 7-198

DIALECT clause, 7-198

IDENTIFIER CHARACTER SET clause,
7-199

keyword interpretation, 7-198, 7-199

KEYWORD RULES clause, 7-199

MODULE keyword, 7-200

multiple transaction declarations, 7-228

NAMES ARE clause, 7-202

names character set, 7-200

national character set, 7-200

NATIONAL CHARACTER SET clause,
7-200, 7-202

QUOTING RULES clause, 7-200

RIGHTS clause, 7-201

SCHEMA keyword, 7-201

specifying date format, 7-198

VIEW UPDATE RULES clause, 7-201

statement-item-name

GET DIAGNOSTICS statement, 7-359

Statement names

in DESCRIBE statement, 7-244

in EXECUTE statement, 7-319

Statement string

in EXECUTE IMMEDIATE, 7-326

Stopping interactive sessions

with EXIT, 7-331

Storage area

assigning row cache, 7-13

CREATE STORAGE AREA clause, 7-11,
7-19

deassigning row cache, 7-17

defining, 7-11

disabling extents, 7-15

enabling extents, 7-15

for compressed data, 7-34

for lists, 7-32

filling randomly, 7-29

filling sequentially, 7-29

setting a default storage area, 7-32

for table rows, 7-32

logical area thresholds, 7-34

page format, 7-17

page size, 7-18

STORAGE MAP clause

of CREATE STORAGE MAP statement, 7-32

Storage maps

CREATE STORAGE MAP statement, 7-25

defining, 7-25

deleting, 7-292

DROP STORAGE MAP statement, 7-292

Stored function

cascade deleting, 7-282

deleting, 7-282, 7-283

restrict deleting, 7-283

STORED NAME IS clause

of CREATE STORAGE MAP statement, 7-34

of CREATE TABLE statement, 7-72

of CREATE TRIGGER statement, 7-113

of CREATE VIEW statement, 7-135

Stored procedure

cascade deleting, 7-282

deleting, 7-271, 7-282, 7-283

- Stored procedure (cont'd)
 - restrict deleting, 7-283
- Stored routine
 - cascade deleting, 7-282
 - deleting, 7-282
 - restrict deleting, 7-283
- STORE IN clause
 - of CREATE STORAGE MAP statement, 7-32
- STORE LISTS clause
 - of CREATE STORAGE MAP statement, 7-32
- STORE RANDOMLY ACROSS clause
 - of CREATE STORAGE MAP statement, 7-34
- STORE USING clause of CREATE STORAGE MAP statement, 7-34
- Storing data
 - in views, 7-136
- Strict partitioning, 7-30
- Summation updates
 - using triggers, 7-109
- SYSPRV privilege, 7-369
- System-defined identifiers, 7-381
- System relations
 - Consult online SQL Help for this information
- System tables
 - Consult online SQL Help for this information

T

- Table
 - ALTER privilege, 7-377
 - cascading delete, 7-296
 - CREATE privilege, 7-377
 - creating, 7-56
 - maximum allowed, 7-80
 - using character set, 7-57
 - DBADM privilege, 7-377
 - DBCTRL privilege, 7-377
 - declaring explicitly, 7-215
 - default protection on, 7-371
 - definitions
 - containing lists, 7-99
 - CREATE TABLE statement, 7-56 to 7-108
 - DROP TABLE statement, 7-296
 - DELETE privilege, 7-377

- Table (cont'd)
 - deleting, 7-296
 - DROP privilege, 7-377
 - global temporary, 7-70
 - granting access, 7-371, 7-392
 - INSERT privilege, 7-377
 - local temporary, 7-70
 - maximum number of, 7-80
 - privileges, 7-371, 7-392
 - REFERENCES privilege, 7-377
 - referencing, 7-69, 7-71
 - SELECT privilege, 7-377
 - SHOW privilege, 7-377
 - specifying
 - in GRANT statement, 7-376, 7-396
 - UPDATE privilege, 7-377
- Table columns
 - data type
 - default character set, 7-57
 - national character set, 7-57
- Table constraints, 7-96
 - See also* CREATE TABLE statement
 - declaring, 7-71
 - in CREATE TABLE statement, 7-64, 7-72
 - in DEFERRABLE clause, 7-64
 - privileges required for declaring, 7-71
- Table cursor
 - See* Cursor
- Table definitions
 - adding to repository, 7-80
- Table-specific constraints
 - See also* CREATE TABLE statement
 - declaring, 7-71
 - privileges required for declaring, 7-71
 - required privileges for, 7-80
 - uses of, 7-79
- Temporary table, 7-70
 - See also* Declared local temporary table
 - deleting rows on commit, 7-71
 - global, 7-70
 - local, 7-70
 - preserving rows on commit, 7-71
- Terminators
 - required for BEGIN DECLARE statement, 7-312

Terminators (cont'd)
 required for END DECLARE statement,
 7-312

THRESHOLDS clause
 of CREATE STORAGE AREA clause, 7-19
 of CREATE STORAGE MAP statement, 7-34

TO clause
 of ANSI/ISO-style GRANT statement, 7-397
 of GRANT statement, 7-380

TRACE statement
 of CREATE TRIGGER statement, 7-110

Transactions, 7-222
 declaring dynamically, 7-228
 ending, 7-249
 environment, 7-223
 format for specifying, 7-223
 in
 context files, 7-223
 DECLARE TRANSACTION statement,
 7-222
 embedded SQL, 7-223
 interactive SQL, 7-223
 isolation levels in, 7-223
 multiple declarations in programs, 7-228
 prestarted
 disabling, 7-150
 restriction in programs, 7-228

Trigger definitions
 adding to repository, 7-80

Triggered action
 defined, 7-110
 specifying with CREATE TRIGGER, 7-113

Triggered statement
 defined, 7-110

Triggers, 7-109
 and external function callouts, 7-117
 and table-specific constraints, 7-79
 cascading deletes, 7-109
 cascading updates, 7-109
 creating, 7-109
 defining a cascading delete trigger, 7-121
 deleting, 7-301
 environment, 7-110
 hidden deletes, 7-109
 in embedded SQL, 7-110, 7-301

Triggers (cont'd)
 in interactive SQL, 7-110, 7-301
 nested, 7-116
 security functions, 7-109
 summation updates, 7-109
 trigger that performs an update, 7-124
 trigger that prevents delete, 7-127

U

UIC identifier
See User identifier

UID identifier
See User identifier

UNION clause
 accessing view containing
 restriction, 7-174

UPDATE clause of CREATE TRIGGER
 statement, 7-113

Update-only cursors, 7-163

UPDATE privilege, 7-377

UPDATE statement
 in CREATE TRIGGER statement, 7-113
 specifying through CREATE TRIGGER
 statement, 7-109
 using with views, 7-136

Updating
 tables
 in CREATE TRIGGER statement, 7-113
 in DECLARE CURSOR statement, 7-163
 views, 7-136

User authentication
 DECLARE ALIAS statement, 7-148
 DROP DATABASE statement, 7-264
 EXPORT statement, 7-333

USER clause
 DECLARE ALIAS statement, 7-151
 DROP DATABASE statement, 7-265
 EXPORT statement, 7-333

User identification code (UIC)
See User identifier

User identifier
 in GRANT statement, 7-381, 7-392

User-supplied name
dynamic SQL statements, 7-244
statement names, 7-244

USING clause
DECLARE ALIAS statement, 7-151
DROP DATABASE statement, 7-265
EXECUTE statement, 7-320
EXPORT statement, 7-333
USING DESCRIPTOR clause
EXECUTE statement, 7-320

V

Value expression
COMPUTED BY columns, 7-64
DEFAULT value, 7-67, 7-207

Variable
specifying dynamic statements, 7-326

Variable declaration
in dynamic SQL, 7-233
in interactive SQL, 7-233

Vertical partitioning, 7-32
VERTICAL PARTITION name clause
CREATE STORAGE MAP statement, 7-35

View
accessing using dynamic cursors
restriction, 7-174
cascading delete, 7-305
changing data in, 7-136

default protection on, 7-371
defining, 7-133 to 7-142
deleting, 7-305
DROP VIEW statement, 7-305 to 7-307
maximum number of, 7-138
read-only, 7-134, 7-136
rules for updating, 7-136
update of
controlling interpretation of
in precompiled SQL, 7-198, 7-201
VIEW UPDATE RULES clause
in precompiled SQL, 7-201
of DECLARE MODULE statement, 7-201

W

WHERE CURRENT OF clause, 7-163
WITH CHECK OPTION clause, 7-134, 7-137,
7-139
WITH GRANT OPTION clause, 7-397
WITH HOLD clause
of DECLARE CURSOR statement, 7-163
WITH LIMIT OF clause of CREATE STORAGE
MAP statement, 7-35
WITH NO EXTENSIONS clause
restriction, 7-335
Write-once, read-many device
storing list information on, 7-43

